Going Higher Than Voevodsky in Category Theory (CT) and Homotopy Type Theory (HoTT) — "Hotter than HoTT"

Douglas C. Youvan

doug@youvan.com

September 1, 2025

This paper proposes a practical way to make higher structure something you can see, run, and check. We couple a visual display functor (Twin Cubes) with a binary tensor truth-table backend that records explicit witnesses for higher cells. Each run yields two synchronized artifacts: a PNG storyboard (a human-readable sequence of mirrored cube states) and a CSV ledger (machine-auditable bits, *_OK flags, and witness IDs). Core operations—faces/degeneracies, composition, inverses, functors, transports—are implemented as sparse Boolean maps. Predicates verify identity, functoriality F(g \circ f) = F(g) \circ F(f), naturality G(f) \circ alpha_A = alpha_B \circ F(f), coherence (triangle and pentagon), and Kan horn fillability up to level n. Equivalences are treated as data: isEquiv(f) is exhibited by (g, eta: id ~ g \circ f, eps: f \circ g ~ id) with concrete witness pointers. Computational univalence is realized via executable ua: Equiv(A,B) -> (A = B) and ua^{-1}, with round-trip checks recorded in the ledger. Rigor is targeted in stages (Soundness, Focused Completeness, Canonicity), and the artifacts interoperate with Cubical Agda, Coq, Lean, and SAT/SMT so independent tools can re-verify results. Pedagogically, Twin Cubes supplies intuition; the tensor truth tables supply auditability. Practically, the artifacts are portable, reproducible, and citable, making "Hotter than HoTT" an engineering blueprint for executable higher structure.

Keywords: Category Theory, Homotopy Type Theory, univalence, computational univalence, equivalence, naturality, coherence, pentagon, triangle, Kan horn, higher-inductive types, pushout, storyboard, truth table, witnesses, reproducibility, Twin Cubes, display functor, verification, SAT, SMT, Cubical Agda, Coq, Lean. 76 pages. A collaboration with GPT-5.

Note on typography: We use an ASCII-safe, LaTeX-style math dialect (e.g., \circ, ua^{-1}, id_A), straight quotes, and snake_case flags. This "CopySafe ASCII Math" avoids Unicode drift, ensuring deterministic copy/paste, auditing, and reproducible hashing across tools.

0. Prelude (Thought Experiment)

Claim
Make higher CT/HoTT operational by coupling a visual display functor (Twin Cubes) to a binary tensor truth-table backend that carries explicit witnesses for higher cells.

Target
Equivalences with witnesses; Kan horn fillability up to level n; computational univalence with ua and ua^{-1} round-trips; coherence laws (triangle, pentagon) as concrete 3+ cell data.

Deliverables (per run)

- Human view: storyboard.png (one frame per meaningful step).
- Machine view: run.csv (rows with bits, op_tag, *_OK flags, *_id witness pointers) and bits.map.csv (stable index map).
- Optional: MANIFEST.json with schema_version, tool versions, and the content hash H = sha256(sigma.json || bits.map.csv || run.csv).

Operational core

- State tensor T over (level, slot, symbol, deco); canonical bitstring s in {0,1}^N via bits.map.csv.
- Operations are sparse Boolean maps:
    - faces and degeneracies: precomputed index permutations,
    - composition: guarded LUT or GF(2) circuit, places g \circ f,
    - inverse: places f^{-1} and expects eta, eps,
    - functor application: block relabel preserving id and \circ,
    - display only: Twin Cubes view is a permutation of slices.
- Display functor is read-only. Use V_render = V \circ m \circ r, where m is a left right mirror involution and r is a rotation from a fixed registry.

Predicates and witnesses

- id_OK, comp_OK, nat_OK, left_inv_OK, right_inv_OK, triangle_OK, pentagon_OK.
- isEquiv_f_OK with columns g_id, eta_id, eps_id.
- ua_OK, ua_inv_OK, ua_rt_OK for univalence and round-trips.

- horn_n_k_OK with filler_id and a rolling closed_up_to_l invariant.
- po_exist_OK, po_unique_OK, po_univ_OK for pushouts.
- Rule: *_OK = 1 only if a well typed witness exists and its ordered boundary matches after normalization. Every green bit stores its *_id, and every red bit stores a reason code.

Rigor gates (staged)

- Soundness: small typed core with deterministic normalization; internal checker re-verifies every boundary before flipping a bit.
- Focused completeness: on finite signatures with a fair scheduler, all required horns up to the chosen level are either filled or permanently reasoned.
- Canonicity: extracted terms reduce to a deterministic TT IR normal form; equalities compare by normal forms plus named higher cells.

Minimal example (reader intuition)

- Choose A, B, and f: A -> B with g: B -> A and witnesses eta: id_A ~ g \circ f, eps: f \circ g ~ id_B.
- Steps: set f and g; add eta and eps; check_equiv -> isEquiv_f_OK = 1; build p = ua(e); build e' = ua^{-1}(p); check round trips -> ua_rt_OK = 1.
- The storyboard shows the same steps on the left and right cubes; the ledger row lists which *_OK bits flipped and which witness IDs were used.

Diagnostics and reasons (examples)

- type_error, slot_occupied, domain_codom_mismatch, inv_missing, eta_missing, eps_missing, nat_witness_absent, triangle_boundary_bad, pentagon_boundary_bad, waiting_on_face, no_filler_found, solver_unsat, solver_model_mismatch, display_collision, display_omission.

What this buys you

- Human readability: Twin Cubes frames you can hand to students.
- Machine audit: CSV you can recheck, diff, and cite.
- Reproducibility: one hash ties PNG and CSV to sigma.json and bits.map.csv.

1. Background and Motivation

1.1 Voevodsky's pillars

- Univalence: types as spaces; equality as paths; ua and ua^{-1} relate equivalences and equalities in U.
- Homotopical foundations: higher paths and coherence data modeled natively; proofs as constructions, not just propositions.
- Proof assistants: Coq/Agda/Lean (and Cubical variants) make derivations precise, mechanized, and shareable.

1.2 Gaps we address

- Operational witnesses: typical papers assert existence; we require a concrete witness with an ID and a boundary that rechecks.
- Machine-auditable infinity-groupoid checks: horns, coherence, and ua round-trips become finite tasks with reason codes and pass/fail bits.
- Portable proofs for pedagogy and reproducibility: every diagram ships with a synchronized duo, storyboard.png + run.csv, tied by a content hash.
- Display vs semantics: the Twin Cubes view is a pure projection (no pixels in the logic loop); math checks run on the ledger, not on the image.
- Small deterministic core: sparse Boolean maps for operations; a tiny TT IR for normalization; a fair scheduler for horns; no green bit without a witness id.
- Interop, not isolation: assistants typecheck emitted witnesses; SAT/SMT proposes candidates that we re-verify before acceptance.

Scope of this paper

- We do not propose a new logic. We specify an execution-and-evidence layer that sits on top of familiar CT/HoTT fragments and produces artifacts that are both human-readable and machine-recheckable.

2. Objects of Study (Signatures and States)

2.1 Signature Sigma (generators and boundaries)

- Purpose: declare the finite fragment we operate on. Sigma lists generators at levels l = 0,1,2,... together with typing/boundary data and optional HIT declarations.

- Level 0 (objects): names like A, B, X, Y.
- Level 1 (morphisms): entries with src, tgt, e.g., f: A -> B, id_A: A -> A.
- Level 2 (paths/homotopies between 1-cells): a named 2-cell with an ordered boundary, e.g., eta: id_A ~ g \circ f, eps: f \circ g ~ id_B.
- Level 3 (coherences between 2-cells): e.g., pentagon: faces [a1, a2, a3, a4, a5] in a fixed order.
- Optional HITs: pushout declarations, spheres, etc., with constructors and equations (glue).
- Well-formedness checks (on load): names unique; sources/targets line up; each higher cell's listed faces type-check and refer to existing lower-level generators.

Minimal Sigma fields (conceptual)

- objects: ["A", "B", ...]
- morphisms: [{name, src, tgt}, ...]
- cells: { "2": [{name, lhs_kind, lhs_args[], rhs_kind, rhs_args[]}, ...],
  "3": [{name, faces[]}, ...] }
- hit: optional block for higher-inductive types (e.g., pushout specs)
- ops: feature switches (compose, inverse, faces, degeneracies) to keep small demos minimal

2.2 State tensor T (one-hot layout)

- Coordinates: (level, slot, symbol, deco).
    - level: 0,1,2,3,...
    - slot: a finite index set per level (e.g., which edge/face cell, or which horn index).
    - symbol: which generator inhabits the slot (e.g., f, g, id_A, eta, pentagon).
    - deco: optional decorations, e.g., cube=L/R for Twin Cubes placement, or registry tags for rotations.
- One-hot invariant: at most one symbol is 1 in a given (level, slot, deco) slice.
- Typing invariant: whenever a composite or boundary is implied by an occupied slice, its source/target must match Sigma's declarations (or the op is rejected with a reason code).

- Sparse implementation: store only 1 positions; all operations act as sparse index permutations or guarded insertions.

2.3 Canonical bitstring s and stable index map

- Flattening: T -> s in {0,1}^N via a deterministic order specified in bits.map.csv.
- bits.map.csv columns (minimal): idx, level, slot_kind, slot_id, symbol_kind, symbol_name, deco_key, deco_val.
- Determinism: sort order is fixed (e.g., level asc, then slot_kind asc, slot_id asc, symbol_kind asc, symbol_name asc, deco lex asc). Once published for a given Sigma version, bits.map.csv is frozen.
- Hash and manifest:
    - H = sha256(sigma.json || bits.map.csv || run.csv).
    - MANIFEST.json records schema_version, filenames, H, and tool versions.
- Display coupling: the Twin Cubes view uses only the projection V_render = V \circ m \circ r applied to slices of s; display does not change s and is not part of any proof predicate.

3. Operations as Sparse Binary Maps

Overview
Each operation transforms the sparse one-hot state T by applying index permutations or guarded insertions. We implement these as total, deterministic maps on the canonical bitstring s in {0,1}^N, with typing checks and reason codes.

3.1 Faces and degeneracies (simplicial style)

- Precompute a sparse permutation/selection matrix P_op for each face/degeneracy op.
- Apply s' = P_op * s over GF(2); because P_op encodes a permutation or copy/drop, the result is still one-hot on each slice.
- Typing guard: every moved bit must land in a slot whose source/target agrees with Sigma. If not, reject with reason type_error.
- Degeneracy policy: mark degenerate slots by deco where needed so later rules can refuse unintended compositions.

3.2 Composition

- Goal: materialize h = g \circ f when tgt(f) = src(g) and the target slot for h is free.
- LUT approach:
    - Lookup key: (f_name, g_name, normalized boundary) -> h_name.
    - Guards: domain_codom_mismatch if tgt(f) != src(g); slot_occupied if h's slot already has a symbol.
    - On success: set the bit for h and record a boundary hash for later equality checks.
- GF(2) circuit approach (optional):
    - Encode composition as a small combinational network that asserts exactly one output wire h is 1 when the pair (f,g) is present and well typed.
    - Use the same guards and reason codes as the LUT version.
- Normalization for later comparisons:
    - Right-associate composites: (h \circ g) \circ f => h \circ (g \circ f).
    - Erase identities on the left or right where allowed by the fragment.
    - Store a normalized boundary string/hash alongside the placed symbol.

3.3 Inverse

- Operation: place f^{-1} into the appropriate slot when requested.
- Guards:
    - Check that f exists and has src/tgt declared in Sigma; type_error if not.
    - slot_occupied if the inverse slot is already filled.
- Witness expectations:
    - Do not auto-green inverse laws. Instead, require named witnesses
        - eta: id_src(f) ~ f^{-1} \circ f
        - eps: f \circ f^{-1} ~ id_tgt(f)
    - The checks left_inv_OK and right_inv_OK turn 1 only when eta and eps are present with matching ordered boundaries.

3.4 Functors

- Abstract action: F maps objects and morphisms, preserves id and \circ.
- Implementation: a block relabel of symbols and slots (deterministic rename) with guards:
    - F(id_A) must land on id_{F(A)}; else set id_OK = 0 with reason image_not_composed or type_error.
    - F(g \circ f) must equal F(g) \circ F(f); the runner materializes both sides and compares normalized boundaries. If equal, set comp_OK = 1 with an optional witness id; else comp_OK = 0 with a reason.
- Transport of 2-cells:
    - A naturality witness alpha_A: F(A) -> G(A) transports to alpha_B under f: A -> B.
    - The nat_OK check compares the two composites G(f) \circ alpha_A and alpha_B \circ F(f) after normalization. A named 2-cell nat_w with that boundary can be recorded; if present and well typed, nat_OK = 1.

3.5 Display functor (Twin Cubes; read-only)

- Map: V_render = V \circ m \circ r where m is a left/right mirror involution and r is a rotation chosen from a fixed registry.
- Behavior: purely a permutation of slices for rendering; never changes s and never affects proof predicates.
- Faithfulness check:
    - Injectivity on support: distinct occupied backend slices must not collide on the same visual slot.
    - Coverage: all intended visual slots for the current frame are covered by V_render(s).
    - display_faithful_OK = 1 iff both conditions hold; otherwise 0 with reason display_collision, display_omission, or bad_mirror.

Step protocol (applies to every op)

1. Read op_tag and arguments.
2. Apply the sparse map or guarded insertion to produce s'.

3. Re-check invariants (typing, one-hot, slot policy). If a guard fails, keep s unchanged and emit a row with *_OK unchanged and a reason code.
4. Evaluate predicates that could have become provable in this step:
    - Collect boundaries, normalize, and either:
      a) mark judgmental equalities directly (no witness id), or
      b) require a named witness whose ordered boundary matches the normalized expression; if so, set *_OK = 1 and store *_id.
5. Write the ledger row t with op_tag, bits_hex_prefix, any *_OK flips, and *_id fields.
6. Render frame t to storyboard.png via V_render.

Determinism and idempotence

- Given (Sigma, bits.map.csv, previous row), the result of an op is fully determined. Re-applying an op that would change nothing is permitted but must not introduce nondeterminism.
- All randomness (if any is later introduced for search) must be seeded and recorded outside this core.

Failure reasons (composition/inverse/functor/display)

- domain_codom_mismatch, slot_occupied, ungrounded_composite, image_not_composed, nat_witness_absent, type_error, display_collision, display_omission, bad_mirror.

4. Predicate Columns with Witnesses

Overview
Predicates are Boolean columns in run.csv that flip to 1 only when there exists a well-typed witness whose ordered boundary matches the normalized goal. Every green bit carries an *_id (or ids) that lets checkers re-verify the claim. Every red bit that was attempted records a reason code.

4.1 Identity (id_OK)

- Goal: F(id_A) = id_{F(A)} or, in plain categories, that an id law holds where expected.
- Method: materialize both sides (or recognize definitional equality) and compare normalized forms.

- Ledger: id_OK in {0,1}. Optional id_witness_id if not judgmental.

4.2 Functoriality (comp_OK)

- Goal: F(g \circ f) = F(g) \circ F(f).
- Method: build both sides, normalize, compare. If equal up to judgment, set comp_OK = 1. If not judgmental, require a named 2-cell witness comp_w.
- Ledger: comp_OK, optional comp_witness_id, or comp_reason if 0.

4.3 Naturality (nat_OK)

- Goal: G(f) \circ alpha_A = alpha_B \circ F(f).
- Method: check types of alpha_A, alpha_B, F(f), G(f); normalize the two routes; accept only with a named 2-cell nat_w or judgmental equality in your fragment.
- Ledger: nat_OK, nat_witness_id if 1, nat_reason if 0.

4.4 Inverses (left_inv_OK, right_inv_OK)

- Goals:
    - left_inv_OK: id_src(f) ~ f^{-1} \circ f via eta.
    - right_inv_OK: f \circ f^{-1} ~ id_tgt(f) via eps.
- Method: require witnesses eta and eps with exact ordered boundaries after normalization.
- Ledger: left_inv_OK, right_inv_OK, eta_id, eps_id. If missing, reasons eta_missing or eps_missing.

4.5 Coherence (triangle_OK, pentagon_OK)

- Goals:
    - triangle_OK: unit-counit triangle law holds (e.g., for an adjunction-like fragment).
    - pentagon_OK: Mac Lane pentagon for associators holds.
- Method: name each 2-face; for pentagon, provide a 3-cell whose boundary is the ordered list of the five 2-cells. Normalize faces before hashing.
- Ledger: triangle_OK, pentagon_OK, triangle_id, pentagon_id, with boundary hashes stored in a witness table or sidecar.

4.6 Equivalence as data (isEquiv_f_OK)

- Goal: isEquiv(f) exhibited by (g, eta, eps) and optional coherence cell(s).
- Method: verify types of g; check eta, eps boundaries; optionally check a minimal coherence if your fragment requires it.
- Ledger: isEquiv_f_OK, g_id, eta_id, eps_id, optional coh_id; isEquiv_reason if 0.

4.7 Computational univalence (ua_OK, ua_inv_OK, ua_rt_OK)

- Goals:
    - ua_OK: construct p = ua(e) for a recorded equivalence e.
    - ua_inv_OK: construct e' = ua^{-1}(p).
    - ua_rt_OK: both round-trips hold: ua^{-1}(ua(e)) = e and ua(ua^{-1}(p)) = p.
- Method: build p and e' as typed objects; record round-trip witnesses rho_e and rho_p or rely on definitional computation where available.
- Ledger: ua_OK (with ua_id), ua_inv_OK (with ua_inv_tuple_id or the component ids), ua_rt_e_OK, ua_rt_p_OK, ua_rt_OK, ua_reason if any 0.

4.8 Kan horns (horn_n_k_OK)

- Goal: for each pending horn Lambda^k[n], record a filler Delta[n] or a permanent reason.
- Method: scheduler enumerates horns; for each, either place a witness filler (constructed or imported from a solver then re-verified) or record a reason such as waiting_on_face or no_filler_found.
- Ledger: horn_n_k_OK, horn_index, n, k, filler_id when 1, horn_reason when 0. Maintain a derived closed_up_to_l invariant once all horns of level <= l are done.

4.9 HIT pushouts (po_exist_OK, po_unique_OK, po_univ_OK)

- Goals:
    - po_exist_OK: for a cone into a pushout P, construct a mediating map u.
    - po_unique_OK: if two mediators exist, produce a path between them.
    - po_univ_OK: bundle existence and uniqueness as the universal property.

- Ledger: po_exist_OK, po_unique_OK, po_univ_OK and ids: po_u_id, po_u_inl_id, po_u_inr_id, po_u_glue_id, po_H_id; po_reason if any 0.

4.10 Display faithfulness (display_faithful_OK)

- Goal: ensure the visual projection is injective on support and covers all intended slots for the frame.
- Ledger: display_faithful_OK in {0,1}, display_reason on failure.

4.11 Witness schema and storage

- Every *_OK = 1 must be backed by at least one witness id:
    - For equalities of 1-cells: a 2-cell id (or judgmental).
    - For pentagon/triangle: the 3-cell id.
    - For horns: a filler id keyed by (n, k, horn_index).
    - For ua: ids for p = ua(e) and e' = ua^{-1}(p), plus rho_e, rho_p if non-definitional.
- Each witness carries:
    - type tag (e.g., 2cell, 3cell, filler_n),
    - ordered boundary serialized in normalized TT IR,
    - boundary_hash = sha256(normalized_boundary_string),
    - optional provenance (internal step index, solver receipt, assistant module name).

4.12 Row protocol for predicates

1. Identify which predicates could flip due to the current op.
2. For each, collect boundary data; normalize to TT IR normal form.
3. If judgmental equality holds in the fragment, set *_OK = 1 (optionally store a trivial witness).
4. Else, look up or allocate a named witness with matching ordered boundary. If found, set *_OK = 1 and write *_id. If not, leave 0 and set a reason code.
5. Never set *_OK = 1 without an id when a witness is required.

4.13 Reason codes (subset used here)

- type_error, domain_codom_mismatch, slot_occupied, image_not_composed, nat_witness_absent, inv_missing, eta_missing, eps_missing, triangle_boundary_bad, pentagon_boundary_bad, waiting_on_face, no_filler_found, solver_unsat, solver_model_mismatch, nf_mismatch, witness_invalidated, display_collision, display_omission, bad_mirror.

Outcome
Predicates become small, checkable contracts tied to concrete witnesses. A reader can see the picture, a machine can re-verify the CSV, and both views point to the same ids and boundary hashes.

5. Equivalences as Data (Not Just Props)

Overview
An equivalence is not a proposition but a record we must exhibit. For a 1-cell f: A -> B, we treat isEquiv(f) as the existence of a tuple (g, eta, eps) with types and boundaries that re-check in the ledger.

5.1 isEquiv(f) record and checks

- Data required
    - g: B -> A
    - eta: id_A ~ g \circ f (left homotopy)
    - eps: f \circ g ~ id_B (right homotopy)
    - optional coherence cells if the chosen fragment demands them
- Typing checks
    - src(g) = B and tgt(g) = A
    - src(eta) = id_A and tgt(eta) = g \circ f
    - src(eps) = f \circ g and tgt(eps) = id_B
- Boundary normalization
    - Normalize both sides of eta and eps (right associate, erase units where allowed).
    - Compute boundary_hash for each 2-cell; use it to dedupe witnesses.

- Success condition
    - isEquiv_f_OK = 1 iff all three pieces are present, well typed, and the ordered boundaries match after normalization.
- Canonicalization
    - If multiple candidates exist for g, pick the one with minimal tuple key (boundary_hash(eta), boundary_hash(eps), local_id).
    - Record tie-breaking in MANIFEST.json if you need stronger determinism.

5.2 Ledger fields

- Required columns
    - isEquiv_f_OK in {0,1}
    - g_id, eta_id, eps_id
- Optional columns
    - isEquiv_reason when 0 (e.g., eta_missing, eps_missing, domain_codom_mismatch, eta_boundary_bad, eps_boundary_bad)
    - coh_id if you store a minimal coherence for inverses in your fragment
    - isEquiv_boundary_hash to stabilize cross-tool linking
    - equiv_tuple_id if you allocate a single id for the whole tuple (in addition to component ids)

5.3 How to build and verify in a run

- Typical step sequence
    1. set_morphism f
    2. set_morphism g
    3. add_witness eta
    4. add_witness eps
    5. check_equiv f,g,eta,eps
- Effects on the ledger
    - The row at step 5 flips isEquiv_f_OK to 1 and writes g_id, eta_id, eps_id.

- If anything is ill typed, the state s is unchanged and isEquiv_reason explains why.

5.4 Interaction with ua

- Link f's equivalence tuple to univalence
  - Build p = ua(e) where e = (f,g,eta,eps). Set ua_OK = 1 with ua_id and a link to the tuple (by hash or id).
  - Build e' = ua^{-1}(p). Set ua_inv_OK = 1 with ua_inv_tuple_id.
  - Check both round-trips; if both hold, set ua_rt_OK = 1 and record rho_e, rho_p if non-definitional.
- Consistency guard
  - If ua(e) exists but e does not pass isEquiv, keep ua_OK = 0 with ua_reason = equiv_missing.

5.5 Transport and functoriality of equivalences

- Image under a functor F
  - From e = (f, g, eta, eps) derive F(e) = (F(f), F(g), F(eta), F(eps)).
  - Re-run the isEquiv check on F(f). If types line up and boundaries normalize to equality, set isEquiv_Ff_OK = 1 (you can either share the same columns or use a namespaced variant).
- Composition of equivalences
  - Given e1 for f: A -> B and e2 for h: B -> C, derive an equivalence for h \circ f by composing the tuples and their witnesses.
  - Record composed witness ids or generate fresh ones; set isEquiv_{h\circ f}_OK = 1 when boundaries re-check.
- Inverse of an equivalence
  - Swap roles: e^{-1} = (g, f, eps, eta). Re-check types and boundaries; set isEquiv_g_OK = 1.

5.6 Display and pedagogy

- Storyboard cues
  - Frame legend lists: isEquiv_f_OK(g=..., eta=..., eps=...).

    - Keep at most 4 flags per frame; if more predicates flip, spill to a keyed legend.
- Physical nets
    - Label the two cubes with f and g faces and draw dotted arcs for eta and eps; the printed RUN.HASH links the haptic demonstration back to the ledger.

5.7 Negative tests and reason codes

- Missing witnesses
    - No eta: isEquiv_f_OK = 0; isEquiv_reason = eta_missing.
    - No eps: isEquiv_f_OK = 0; isEquiv_reason = eps_missing.
- Typing and boundary errors
    - g has wrong type: isEquiv_f_OK = 0; isEquiv_reason = type_error.
    - Domain/codomain mismatch in composing g \circ f or f \circ g: isEquiv_f_OK = 0; isEquiv_reason = domain_codom_mismatch.
    - Boundaries do not normalize to the required forms: isEquiv_f_OK = 0; isEquiv_reason = eta_boundary_bad or eps_boundary_bad.
- Invalidation
    - If later edits overwrite f or g or change a boundary, flip isEquiv_f_OK back to 0 and set isEquiv_reason = witness_invalidated.

5.8 Minimal working example

- Sigma
    - Objects: A, B
    - Morphisms: id_A, id_B, f: A -> B, g: B -> A
    - 2-cells: eta: id_A ~ g \circ f, eps: f \circ g ~ id_B
- Run
    - set_morphism f; set_morphism g; add_witness eta; add_witness eps; check_equiv f,g,eta,eps; ua_from_equiv; ua_inv_from_path; ua_roundtrip
- Expected flips
    - isEquiv_f_OK = 1; ua_OK = 1; ua_inv_OK = 1; ua_rt_OK = 1

Outcome
Equivalences become concrete artifacts with ids and boundary hashes. They transport through functors, compose, invert, and power univalence, all recorded as small, re-checkable facts in the ledger.

6. Computational Univalence

Overview
Univalence is executed, not only asserted. We represent both directions, ua and ua^{-1}, as concrete constructors operating on equivalence tuples and universe-level paths. Round-trips are checked and recorded in the ledger.

6.1 Representation of paths in the universe U

- PathU term: a value p : (A = B) in U is represented by a small TT-IR node PathU(Ua e) or SymU(name) with an ordered boundary (A,B).
- Equivalence tuple: e = (f, g, eta, eps) with required types and 2-cell boundaries.
- Boundary function:
    - src_U(p) = A, tgt_U(p) = B
    - src(e) = A, tgt(e) = B (induced from f : A -> B and g : B -> A)

6.2 ua(e) : Equiv(A,B) -> (A = B)

- Precondition: isEquiv_f_OK = 1 for the tuple e; else reject with ua_reason = equiv_missing.
- Construction: p := ua(e). Record ua_id and link to the tuple (equiv_tuple_id or boundary hash).
- Ledger: set ua_OK = 1 with ua_id when p is placed and typed.

6.3 ua^{-1}(p) : (A = B) -> Equiv(A,B)

- Precondition: p typed with src_U(p) = A, tgt_U(p) = B.
- Construction: e' := ua^{-1}(p) = (f', g', eta', eps') as a concrete tuple in the ledger.
- Ledger: set ua_inv_OK = 1 with ua_inv_tuple_id and component ids f'_id, g'_id, eta'_id, eps'_id.

6.4 Round-trips and linkage

- Check 1: ua^{-1}(ua(e)) = e
    - Build e1 := ua^{-1}(ua(e)).
    - Compare e1 with e componentwise via normalization. If judgmental, mark ua_rt_e_OK = 1; else require witnesses rho_e for each component equality.
- Check 2: ua(ua^{-1}(p)) = p
    - Build p1 := ua(ua^{-1}(p)).
    - Compare normalized PathU forms. If judgmental, set ua_rt_p_OK = 1; else require a path witness rho_p.
- Aggregate: ua_rt_OK = 1 iff both ua_rt_e_OK and ua_rt_p_OK are 1.
- Ledger fields: ua_id, ua_inv_tuple_id, ua_rt_e_OK, ua_rt_p_OK, ua_rt_OK, optional rho_e_id, rho_p_id, ua_reason if any 0.

6.5 Normalization and equality policy (TT-IR)

- 1-cells: right-associate compositions; erase identities as permitted.
- PathU nodes: normalize to Ua(e_hash) where e_hash is the canonical hash of the tuple (boundary-based), so equal ua images compare structurally.
- Equivalence tuples: compare by normalized components; when not judgmental, require 2-cell witnesses per component.

6.6 Failure modes and reasons

- equiv_missing: ua called before isEquiv_f_OK.
- type_error: p has inconsistent src/tgt; or tuple components ill typed.
- link_mismatch: ua path does not link to the claimed tuple (hash mismatch).
- transport_fail: attempted transport along p failed typing checks in the chosen fragment (if transports are enabled).
- nf_mismatch: post-round-trip components fail to normalize to the originals and no witnesses are provided.

6.7 Micro-run (typical)

- Steps: build e = (f,g,eta,eps) -> isEquiv_f_OK; call ua(e) -> ua_OK; call ua^{-1}(p) -> ua_inv_OK; run ua_roundtrip -> ua_rt_OK.
- Expected: ua_OK = 1; ua_inv_OK = 1; ua_rt_OK = 1. If any precondition fails, the op is rejected and ua_reason explains why.

6.8 Interop (assistants and solvers)

- Assistants: codegen a module with constants for e, p, and the round-trip lemmas mirroring ua_rt_* columns; typecheck receipts are recorded externally.
- Solvers: optional side-conditions (e.g., equalities between normalized composites used inside tuples) may be emitted as SMT/CNF; any model must be re-imported and re-verified before flipping bits.

Outcome
Univalence becomes a pair of executable maps tied to concrete data. The ledger records not just that ua and ua^{-1} exist, but that both directions compose back to the original objects, with either judgmental agreement or explicit witnesses.

7. Kan Horn Fillability (Toward infinity-groupoids)

Overview
We operationalize horns as finite tasks. Each pending horn Lambda^k[n] is registered with a canonical boundary hash and either gets a filler Delta[n] or a permanent reason code. A fair scheduler attempts horns by level bands so closure up to level n becomes a concrete, checkable invariant.

7.1 Shapes and slots

- Horn shape: Lambda^k[n] is an n-simplex boundary with the k-th face missing.
- Filler: Delta[n] with an ordered list of (n-1)-faces.
- Slots: allocate horn(n,k,i) and filler(n,i) slots indexed by a stable horn_index i.
- Typing: all present faces must be well typed and share a consistent boundary.

7.2 Discovery and registration

- Detect partial boundaries created by prior steps (composites, transports, 2-cells, etc.).
- Normalize each listed face to TT IR normal form.

- Compute boundary_hash = sha256(serialized_ordered_faces).
- If this boundary_hash is new, register horn Lambda^k[n] as pending with a fresh horn_index; else reuse the existing index.

7.3 Scheduler (fair, banded)

- Maintain bands B_1, B_2, ..., B_n where B_m holds all pending horns of level m.
- Round-robin order: iterate B_1..B_n, attempting the next K_m horns from each band per tick (K_m is a small capacity per level).
- Per-horn counters: attempts, last_attempt_step, backoff.
- Starvation guard: if a horn has not been attempted in R rounds, force an attempt or mark scheduler_starvation (diagnostic only).

7.4 Filling strategies

- Deterministic constructors: apply known introduction rules for low n (e.g., compose 2-cells, whisker, unitors) to propose a filler.
- Bounded search: enumerate a small space of candidate composites consistent with the boundary; stop at first boundary match.
- External solvers: export CNF or SMT for the side condition; on SAT, re-import the model and reconstruct a concrete filler, then re-verify boundaries internally. Never flip bits based only on solver output.
- Verification: check that the constructed Delta[n] has exactly the required ordered faces after normalization; if so, allocate filler_id.

7.5 Ledger schema

- Columns: horn_n_k_OK in {0,1}, horn_index, n, k, filler_id (if 1), horn_reason (if 0).
- Derived invariants: closed_up_to_l = 1 if all horns with level <= l are either 1 or carry a permanent reason not caused by scheduler limits. Emit a summary line or a manifest field for each l.

7.6 Canonical choice and dedupe

- If multiple fillers exist, pick the one with minimal (boundary_hash_of_faces, constructor_tag, local_id).

- If a later attempt proposes a different filler for the same boundary, reject and log duplicate_boundary (diagnostic) or accept only if your policy allows replacing with a strictly smaller canonical choice.

7.7 Interactions with other predicates

- Filling a 2-horn may immediately certify comp_OK, nat_OK, or left/right_inv_OK for the same boundary. Re-evaluate affected predicates in the same step.
- Filler witnesses can serve as faces of higher horns; registration should happen after each successful fill.

7.8 Progress guarantees (focused completeness)

- On finite Sigma and a fixed n, the number of distinct horn boundaries encountered during a bounded run is finite.
- With fair scheduling and capacity K_m > 0 for all m <= n, each pending horn is either filled, proved impossible in the chosen fragment, or marked with a permanent reason such as type_error or noncanonical_boundary.
- CI gate: closed_up_to_2 must be 1 for M2 demo suites; higher l as milestones increase.

7.9 Examples

- n = 2, k = 1 (square with one edge missing): boundaries come from two parallel composites; filler is a 2-cell witnessing their equality.
- n = 3, k arbitrary: boundaries are five 2-cells; a 3-cell filler witnesses a coherence like the pentagon.
- Naturality as horn: G(f) \circ alpha_A and alpha_B \circ F(f) define a 2-horn; a naturality 2-cell fills it.

7.10 Display cues (Twin Cubes)

- Pending horns: small hollow markers near the missing face; index label i.
- Filled horns: solid markers with filler_id.
- Keep at most 4 markers per frame; overflow goes to a keyed legend.

7.11 Engineering notes and reasons

- Storage: keep a horn registry keyed by boundary_hash with fields {n,k,faces,attempts,status}.

- Capacity: choose K_2 >= 32 and K_3 >= 8 for demo scale; expose as config.
- Performance: cache normalized faces; reuse fillers across runs via content-addressed storage.
- Common reasons: waiting_on_face, type_error, noncanonical_boundary, no_filler_found, solver_unsat, solver_model_mismatch, perf_budget_exceeded (warning), scheduler_starvation (diagnostic).

Outcome
Kan conditions become concrete, finite work items with IDs and reasons. Closure up to level n is not a slogan but a row-level fact you can check in run.csv and visualize on the storyboard, with every accepted filler re-verified by the same small normalizer that governs the rest of the system.

8. Higher-Inductive Types (HITs) and Pushouts

Overview
We treat core HITs as first-class generators with constructors and equations that become witnesses in the ledger. The flagship example is a pushout, encoded so its universal property appears as small, checkable bits with explicit witness ids.

8.1 Generators (pushout example)

- Data: a span X -(f)-> Y and X -(g)-> Z.
- New type: P := Pushout(f,g).
- Point constructors (1-cells):
    - inl : Y -> P
    - inr : Z -> P
- Path constructor (2-cell family):
    - glue_x : inl \circ f x ~ inr \circ g x for each x : X
- In Sigma: declare P, inl, inr as level-1 generators; declare a family glue with ordered boundary [inl \circ f, inr \circ g].

8.2 Typing and slots

- Allocate slots for P, inl, inr, and for each glue_x (practically: for a finite set of x in demos, or a schematic family with parameters tracked in deco).

- One-hot and typing guards ensure inl, inr, and every glue face are well typed relative to Sigma.

8.3 Universal property as predicates

- Given a cocone into Q consisting of u : Y -> Q, v : Z -> Q, and a witness h_x : u \circ f x ~ v \circ g x (compatible with glue), there exists a unique mediator m : P -> Q satisfying:
    - m \circ inl = u
    - m \circ inr = v
    - transport of glue_x along m matches h_x
- We encode three predicates:
    - po_exist_OK = 1 when a mediator m with the required boundaries is constructed and verified.
    - po_unique_OK = 1 when any two mediators m1, m2 are witnessed equal by a path H (homotopy).
    - po_univ_OK = 1 when both existence and uniqueness are established for the given cocone.
- Ledger ids: po_u_id (mediator), po_u_inl_id, po_u_inr_id, po_u_glue_id (the equation witnesses for inl, inr, glue), and po_H_id for uniqueness when present.

8.4 Construction steps (typical run)

1. Declare the span objects and maps X, Y, Z, f, g.
2. Introduce P = Pushout(f,g), and constructors inl, inr.
3. Add glue_x witnesses for the selected x values (or a schematic family marker in deco).
4. Provide a cocone (u, v, h_x) into Q.
5. Invoke make_pushout_mediator to construct m : P -> Q; re-verify boundaries; set po_exist_OK = 1 and record ids.
6. If two mediators arise (m1, m2), record a 2-cell H : m1 ~ m2 and set po_unique_OK = 1.
7. Set po_univ_OK = 1 when both bits are 1 for this cocone.

8.5 Normalization and hashes

- Normalize the equalities m \circ inl = u and m \circ inr = v using the same TT IR as elsewhere (right-associate, erase units).
- For each glue_x, normalize both sides of transported equations before hashing.
- Store boundary_hash for each witness so external tools can re-check.

8.6 Interaction with horns

- The equation m \circ inl = u is a 2-horn whose filler is the witness you record; same for m \circ inr = v and for transported glue_x.
- Therefore, setting po_exist_OK typically flips a small set of horn_n_k_OK bits as well.

8.7 Functoriality of pushouts (optional, if F is declared)

- If a functor F is available, export a mapping from Pushout(f,g) to Pushout(F f, F g) and verify that inl, inr, glue transport appropriately.
- Record functoriality via comp_OK and nat_OK bits on the images; if you prove the canonical comparison is an equivalence, set isEquiv_OK on that comparison map.

8.8 Failure modes and reasons

- type_error: inl/inr/glue boundaries do not type-check against Sigma.
- bad_cospan: the span (f,g) is malformed or uses undeclared objects.
- glue_boundary_bad: a provided glue_x does not normalize to inl \circ f x and inr \circ g x.
- u_missing: missing u or v for the cocone.
- u_inl_bad / u_inr_bad: mediator equations to inl or inr fail to verify.
- no_mediator_found: construction or search could not build m within limits.
- po_reason holds one of these strings when any po_*_OK = 0.

8.9 Minimal pushout demo (finite case)

- Sigma: choose finite X, Y, Z with explicit f, g on enumerated elements; declare P, inl, inr, glue_x for all x in X.
- Cocone: pick Q and maps u, v and h_x per x.

- Run: introduce P, add u, v, h_x, call make_pushout_mediator, verify, then (optionally) show uniqueness by building H between two mediators.
- Expected flips: po_exist_OK = 1; po_univ_OK = 1 (if uniqueness is shown); several horn_n_k_OK = 1 for the associated equalities.

Outcome
Pushouts (and by extension other HITs) are encoded so their universal property is just a handful of small, explicit witnesses with ids and hashes. Readers see the constructors and equations on Twin Cubes; machines re-check the same facts in run.csv, closing the gap between intuition and audit.

9. Twin Cubes Interface (Human View)

Overview
Twin Cubes is the human-facing view. It renders each ledger step as a frame showing two synchronized cubes (L and R). The view is a pure projection; it never affects math predicates. A reader should be able to follow the proof in pictures, and the same run is auditable in run.csv.

9.1 View map and faithfulness

- Rendering map: V_render = V \circ m \circ r
    - V: base placement from backend slots to visual slots.
    - m: left/right mirror involution (L <-> R).
    - r: rotation from a fixed registry (e.g., 24 cube rotations).
- Faithfulness check:
    - Injectivity on support: no two occupied backend slots map to the same visual slot.
    - Coverage: all intended visual slots for this frame are populated.
    - display_faithful_OK = 1 iff both hold; else record display_reason in {display_collision, display_omission, bad_mirror}.

9.2 Storyboard.png specification

- One frame per ledger row t that changes something meaningful.
- Frame layout:

    - Left panel: cube L; Right panel: cube R.
    - Caption line: "t=<index> op_tag=<name> H=<hash prefix>".
    - Legend: list only the flags that flipped at t (max 4 lines). Example:
        - isEquiv_f_OK(g=12, eta=34, eps=35)
        - ua_rt_OK
- Glyph policy:
    - Face labels from symbol_name (shortened if needed).
    - At most 12 glyphs per cube; if more, spill to keyed legend.
- Determinism:
    - Same (Sigma, bits.map.csv, run.csv) must produce byte-identical PNGs.
    - Renderer version recorded in a small footer: "renderer=vX.Y H_run=sha256:...".

9.3 Physical nets (printable)

- Two printable nets: L and R, each with the same RUN.HASH.
- Face order and gluing tabs fixed per template; fonts 12pt or larger.
- Optional QR code payload:
    - {"hash":"sha256:...","sigma":"sigma.json","map":"bits.map.csv","run":"run.csv"}
- Use the same glyphs as the PNG to align classroom demos with the ledger.

9.4 What changes per frame

- Show only differences from t-1:
    - Newly placed symbols (highlight for this frame only).
    - Predicates that flipped to 1 (tick marks near relevant faces/edges).
    - Pending horns and filled horns markers (see 9.7).
- Do not re-animate static context; keep visual noise low.

9.5 Legends and density limits

- Max 4 predicate lines in the on-frame legend. If more predicates flip, add a keyed legend page after the frame or show a compact summary: "+3 more (see ledger)".

- Keep each line short: name(args) or name=1; include witness ids only when helpful.

9.6 Accessibility and color policy

- No meaning by color alone. Shape and position carry semantics.
- Contrast suitable for printing and projection. Keep a linter that checks basic contrast thresholds.
- Always provide alt text for each frame, e.g., "t=7: placed eta on L-face; isEquiv_f_OK flipped with g=12, eta=34, eps=35."

9.7 Horn markers

- Pending horn: hollow marker near target face; label "horn n=<n> k=<k> i=<index>".
- Filled horn: solid marker with "filler_id=<id>".
- Limit markers to 4 per frame; overflow goes to the legend.

9.8 Frame selection policy

- Emit a frame when:
    - s changes at a visible slot, or
    - any *_OK bit flips, or
    - display_faithful_OK flips.
- Do not emit frames for no-ops or internal housekeeping. The ledger may still record these rows; the viewer filters them out.

9.9 File naming and run coupling

- storyboard.png is tied to H = sha256(sigma.json || bits.map.csv || run.csv).
- The footer includes H; if any input changes, H changes and PNG bytes change.
- Optional: storyboard_debug.png includes extra overlays (slot indices, bounding boxes) for development, but do not publish it.

9.10 Error overlays (optional, when display_faithful_OK=0)

- If a frame fails faithfulness, still render it with a red "Display not faithful" banner and show display_reason.
- This does not affect math. The ledger remains the source of truth; the banner prevents readers from misinterpreting the picture.

Outcome
Twin Cubes gives a compact, deterministic picture per step while staying out of the proof loop. The same witness ids that make bits flip in run.csv appear in the frame legend and printed nets, so human intuition and machine audit remain synchronized.

10. File and Run Artifacts (Machine View)

Overview
This section specifies the file set that makes a run auditable and reproducible. The core contract is: given the trio (sigma.json, bits.map.csv, run.csv), a verifier must be able to re-check every green bit and regenerate storyboard.png byte-identically.

10.1 run.csv (row-level ledger)
Purpose
Append-only log of steps and checks. Each row represents one applied op (or a no-op with a reason). Do not rewrite past rows; instead append corrective steps.

Required columns (order is fixed once published)
t, op_tag, bits_hex_prefix, id_OK, comp_OK, nat_OK, left_inv_OK, right_inv_OK, triangle_OK, pentagon_OK, isEquiv_f_OK, g_id, eta_id, eps_id, isEquiv_reason, ua_OK, ua_id, ua_inv_OK, ua_inv_tuple_id, ua_rt_e_OK, ua_rt_p_OK, ua_rt_OK, ua_reason, horn_n_k_OK, horn_index, n, k, filler_id, horn_reason, po_exist_OK, po_unique_OK, po_univ_OK, po_u_id, po_u_inl_id, po_u_inr_id, po_u_glue_id, po_H_id, po_reason, display_faithful_OK, display_reason

Notes

- t: monotone integer index starting at 0.
- op_tag: short verb_noun, e.g., set_morphism, add_witness, check_equiv, ua_from_equiv, ua_inv_from_path, make_pushout_mediator, export_cnf, import_sat_horn.
- bits_hex_prefix: first 32 hex chars of the canonical bitstring s for quick diffing; optional full hash may appear in MANIFEST.json.
- *_OK: 0 or 1. Never set 1 without the corresponding *_id when a witness is required.
- *_id fields: stable integers or content-addressed strings. Use empty string when not set.
- *_reason fields: one of the cataloged reason codes when the related *_OK is 0 due to an attempted check.

Integrity lints (verifier should enforce)

- Column set and order exactly match the declared schema for the paper.
- Every row has op_tag set; unknown tags are allowed but must not flip bits that require witnesses unless ids are present.
- If any *_OK == 1 and its required *_id is empty, fail the ledger.
- If horn_n_k_OK == 1 then filler_id must be non-empty and (n,k,horn_index) must be consistent across the file.

10.2 bits.map.csv (stable index map)
Purpose
A frozen mapping from tensor coordinates to bit indices that flattens T to s in {0,1}^N.

Required columns (deterministic order)
idx, level, slot_kind, slot_id, symbol_kind, symbol_name, deco_key, deco_val

Notes

- idx: 0-based integer index into s.
- level, slot_kind, slot_id: identify the slot (e.g., level=1, slot_kind=edge, slot_id=17).
- symbol_kind, symbol_name: what occupies the slot when the bit is 1 (e.g., morphism, f).
- deco_key, deco_val: optional decorations (e.g., cube=L or R). Use empty string if not present.
- Freeze policy: once a version of Sigma is published, the bits.map.csv for that Sigma version is immutable. If you must change ordering, bump the Sigma version and regenerate all artifacts with a new hash.

10.3 CNF/SMT export (optional side conditions)
Purpose
Encode selected side conditions (e.g., horn choices, naturality equalities) for an external solver to propose candidates. Solver outputs must be re-imported and re-verified internally before flipping any *_OK bit.

Conventions

- CNF files use DIMACS with one-hot encodings; solver models go to .sol.
- SMT-LIB files end with (check-sat)(get-model); models go to .model.

- Every export action appends a row in run.csv with op_tag=export_cnf or export_smt and a short note in a sidecar log if desired.
- Every import action appends a row with op_tag=import_sat_horn or import_smt_horn and sets horn_n_k_OK only after internal verification succeeds; otherwise writes horn_reason.

10.4 Dense state store (optional)
Purpose
Speed and scale. The canonical bitstring s and selected columns can be mirrored to a columnar store (e.g., Parquet). This is not part of the minimal artifact contract.

Rules

- If you publish Parquet, include its content hash in MANIFEST.json optional[] and ensure a verifier can reconstruct the same Parquet from the CSVs.

10.5 MANIFEST.json (run manifest)
Purpose
Tie the artifacts together and record tool versions and the content hash H.

Minimal fields
schema_version: "tt-ledger/1.0"
sigma_file: "sigma.json"
bits_map_file: "bits.map.csv"
run_file: "run.csv"
png_file: "storyboard.png"
hash: "sha256:..." (H = sha256(sigma.json || bits.map.csv || run.csv))
tool_versions: { "tt-runner": "x.y.z", "renderer": "a.b.c" }
created_utc: "2025-08-31T23:59:59Z"

Optional fields
optional: ["out/state.parquet", "out/horns/*.cnf", "out/horns/*.model"]
expressivity: { "max_level": 3, "hit": ["pushout"], "ua": true }
environment: { "python": "3.11.6", "platform": "Windows 11" }

10.6 Regeneration and determinism contract
Given the tuple (sigma.json, bits.map.csv, run.csv):

- A checker must be able to:
    - recompute every predicate from scratch and agree with all *_OK bits,

    - verify that each *_OK == 1 has a matching *_id with a boundary that normalizes to the recorded goal,
    - confirm referential integrity: each id used actually appears in the internal witness table.
- A renderer must produce a byte-identical storyboard.png. The renderer must not read any files other than the trio above (and optionally MANIFEST.json for versions).
- The combined content hash H must change if and only if any of the trio changes. H must appear in the PNG footer and, optionally, printed on physical nets.

10.7 Naming and ids

- Use numeric ids for small demos (e.g., 12, 34, 35). For larger sets, prefer content-addressed ids: hex-encoded sha256 of the normalized boundary string, optionally truncated (e.g., w_3f29c1).
- For tuples, either store component ids only or also store an equiv_tuple_id that is a stable function of the component hashes.
- Do not reuse ids across logically different witnesses. If a witness is invalidated (e.g., its faces changed), allocate a new id and mark witness_invalidated in the next row that depends on it.

10.8 Reasons and audit strings

- All reason codes come from a fixed catalog (see appendix). Never invent ad hoc strings in run.csv; if you need a new code, add it to the catalog table in the paper and bump schema_version.
- Keep free-form notes (if any) in a separate sidecar log; do not pollute run.csv reasons.

10.9 Quick minimal headers (for reference)
bits.map.csv
idx,level,slot_kind,slot_id,symbol_kind,symbol_name,deco_key,deco_val

run.csv
t,op_tag,bits_hex_prefix,id_OK,comp_OK,nat_OK,left_inv_OK,right_inv_OK,triangle_OK,pentagon_OK,isEquiv_f_OK,g_id,eta_id,eps_id,isEquiv_reason,ua_OK,ua_id,ua_inv_OK,ua_inv_tuple_id,ua_rt_e_OK,ua_rt_p_OK,ua_rt_OK,ua_reason,horn_n_k_OK,horn_index,n,k,filler_id,horn_reason,po_exist_OK,po_unique_OK,po_univ_OK,po_u_id,po_u_inl_id,po_u_inr_id,po_u_glue_id,po_H_id,po_reason,display_faithful_OK,display_reason

Outcome
The machine-view artifacts form a compact, deterministic contract: a verifier can re-derive every check bit and a renderer can regenerate the identical PNG. Because every green bit carries an id and a boundary hash, external tools can independently confirm the same facts without trusting our code, and any change anywhere propagates to the single content hash H.

11. Milestones (M0–M6)

Overview
Each milestone adds one visible capability and one audit capability. Every milestone must pass CI gates: determinism, referential integrity, no green bit without an id, and reproducible storyboard.png from the trio (sigma.json, bits.map.csv, run.csv).

M0 — Encoder / Exporter (plumbing)
Goal

- Load Sigma, build bits.map.csv, flatten T to s, write run.csv, render storyboard.png deterministically.

Scope

- ops: set_morphism, add_witness, no-ops with reasons.
- renderer: V_render = V \circ m \circ r; display_faithful_OK.
- hashing: H = sha256(sigma.json || bits.map.csv || run.csv); MANIFEST.json.

Acceptance

- Deterministic re-run yields byte-identical storyboard.png and same H.
- Lints: column order fixed; ids present whenever *_OK = 1.

CI gates

- Golden artifacts for a tiny Sigma (A,B,f,g,eta,eps).
- Ledger lint: no *_OK = 1 without an accompanying *_id where required.

M1 — Equivalences + basic ua
Goal

- Encode isEquiv(f) = (g, eta, eps). Execute ua and ua^{-1}. Record round-trips.

Scope

- Predicates: isEquiv_f_OK, ua_OK, ua_inv_OK, ua_rt_e_OK, ua_rt_p_OK, ua_rt_OK.
- TT-IR: normalization for 1-cells; tuple comparison.

Acceptance

- Minimal demo passes: isEquiv_f_OK = 1; ua_OK = 1; ua_inv_OK = 1; ua_rt_OK = 1.
- Negative tests: missing eta/eps produce isEquiv_reason; ua before isEquiv produces ua_reason = equiv_missing.

CI gates

- Re-derive all ua bits from ledger; verify witness ids link to correct boundaries.

M2 — Kan up to level 2 (closed_up_to_2)
Goal

- Register and fill 2-horns (squares). Provide scheduler, exports, and re-import with internal verification.

Scope

- horn_n_k_OK with (n=2, any k); filler_id; horn_reason.
- Export: CNF/SMT one-hot; Import: DIMACS/SMT model re-import; internal re-check.
- Derived invariant: closed_up_to_2.

Acceptance

- Demo suite closes all level-2 horns or gives permanent reasons.
- Re-imported solver models never flip bits without internal verification.

CI gates

- closed_up_to_2 = 1 on demos.
- All import rows contain filler_id on success, horn_reason on failure.

M3 — Coherence (triangle, pentagon)
Goal

- Record triangle_OK and pentagon_OK with explicit 3-cell witnesses.

Scope

- 2-cell face normalization; 3-cell boundary serialization and hashing.
- Viewer: minimal overlay to indicate coherence witnesses.

Acceptance

- Associator pentagon on Fin(3) passes with a named 3-cell; triangle for unitors passes.
- Negative tests catch triangle_boundary_bad or pentagon_boundary_bad.

CI gates

- Re-check that the 3-cell boundary matches the ordered list of 2-cells.

M4 — HITs: Pushouts

Goal

- Encode Pushout(f,g) with constructors inl, inr, glue_x. Prove the universal property on finite demos.

Scope

- Predicates: po_exist_OK, po_unique_OK, po_univ_OK; ids: po_u_id, po_u_inl_id, po_u_inr_id, po_u_glue_id, po_H_id.
- Interactions: horn fills for the mediator equalities.
- Optional: functorial transport of pushouts.

Acceptance

- Finite-span demo: mediator constructed and verified; po_univ_OK = 1.
- Failure cases produce po_reason (bad_cospan, glue_boundary_bad, u_inl_bad, u_inr_bad, no_mediator_found).

CI gates

- All mediator equalities re-check from TT-IR; witness ids non-empty when *_OK = 1.

M5 — Interop (Assistants and Solvers)

Goal

- Round-trip with Cubical Agda/Coq/Lean; stable CNF/SMT export and model re-import; receipts recorded in MANIFEST optional[].

Scope

- Codegen: assistant stubs that reference witness ids; optional typecheck receipts.
- Export/import: directory policies, naming, and receipts.

Acceptance

- At least one demo typechecks in an assistant; receipts recorded.
- SAT/SMT loops complete on demos with successful re-import and internal verification.

CI gates

- Tool version pins; receipts present and match ledger hashes.

M6 — Raise n and checkpoint colimits
Goal

- Extend horns to level 3 (selected 3-horns) with scheduler fairness.
- Add at least one more HIT or colimit (e.g., coequalizer) with a minimal universal property check.

Scope

- horn_n_k_OK for n=3; closed_up_to_3 on demos.
- New HIT predicate columns mirrored after pushout pattern.

Acceptance

- Demo closes selected 3-horns or provides permanent reasons; closed_up_to_3 = 1 for that subset.
- New HIT passes existence/uniqueness checks on a finite instance.

CI gates

- Performance envelope documented (max rows, time per op) on your Beelink box.
- Determinism preserved across all new ops and witnesses.

Common risk controls across milestones

- Fail closed: do not flip *_OK without ids.
- Referential integrity: every id referenced must exist in the witness table.
- Determinism: re-run must reproduce storyboard.png bytes and H exactly.

- Reason catalog: all failure paths mapped to a fixed, documented code.

12. Flagship Demo Cases

Overview
Three compact demos exercise the full pipeline: equivalences and ua on finite sets, a pentagon coherence on Fin(3), and a naturality square transported by a functor. Each ships with Sigma, bits.map.csv, run.csv, storyboard.png, and MANIFEST.json. All math is re-checkable from the trio (sigma.json, bits.map.csv, run.csv).

12.1 Finite sets: explicit equivalence and ua round-trip
Goal

- Exhibit isEquiv(f) for a bijection f: Fin(n) -> Fin(n), compute p = ua(e), recover e' = ua^{-1}(p), and check both round-trips.

Sigma (conceptual)

- objects: A = Fin(n)
- morphisms: id_A, f: A -> A, g: A -> A
- 2-cells: eta: id_A ~ g \circ f, eps: f \circ g ~ id_A

Run outline

- set_morphism f; set_morphism g; add_witness eta; add_witness eps
- check_equiv f,g,eta,eps
- ua_from_equiv e
- ua_inv_from_path p
- ua_roundtrip

Expected ledger flips

- isEquiv_f_OK = 1; ua_OK = 1; ua_inv_OK = 1; ua_rt_e_OK = 1; ua_rt_p_OK = 1; ua_rt_OK = 1

Storyboard cues

- Frame shows L/R cubes with f and g on corresponding faces; a small legend lists flipped bits and ids (g_id, eta_id, eps_id, ua_id).

12.2 Associator pentagon on Fin(3)

Goal

- Display the Mac Lane pentagon coherence using permutations on Fin(3) (or any small strict monoidal proxy), record a 3-cell witness with ordered 2-face boundary.

Sigma (conceptual)

- objects: A, B, C corresponding to 3 labeled items
- morphisms: generators for associators alpha_{-,-,-} as needed
- 2-cells: the five 2-face composites around the pentagon
- 3-cell: pentagon with faces [f1, f2, f3, f4, f5] in fixed order

Run outline

- materialize the five 2-face composites by composition steps
- add_witness pentagon_3cell with the ordered boundary
- check_coherence pentagon

Expected ledger flips

- pentagon_OK = 1 with pentagon_id and boundary_hash recorded

Storyboard cues

- A single frame highlighting the five faces and a badge "pentagon_OK (id=...)" in the legend.

12.3 Naturality square transported under a functor

Goal

- For a square G(f) \circ alpha_A = alpha_B \circ F(f), record a 2-cell naturality witness and set nat_OK.

Sigma (conceptual)

- objects: A, B
- morphisms: f: A -> B; functor images F(f), G(f)
- 2-cells: alpha_A: F(A) -> G(A), alpha_B: F(B) -> G(B), nat_w: G(f) \circ alpha_A ~ alpha_B \circ F(f)

Run outline

- set_morphism f; apply functors F and G (block relabel)
- add_witness alpha_A, alpha_B
- add_witness nat_w with the ordered boundary
- check_naturality f, alpha_A, alpha_B

Expected ledger flips

- comp_OK may flip during functor checks; nat_OK = 1 with nat_witness_id

Storyboard cues

- Two arrows on L/R indicate the alternative routes; legend: "nat_OK (w=...)".

12.4 Packaging and reproducibility

- Include all four files: sigma.json, bits.map.csv, run.csv, storyboard.png, plus MANIFEST.json with hash H.
- Verify determinism: rerun the renderer to produce byte-identical storyboard.png.
- Provide a short README that lists exact op_tag sequences used in the run.

12.5 Classroom kit (physical nets)

- Print L/R nets labeled with the same RUN.HASH; small sticky labels for witness ids (eta_id, eps_id, pentagon_id).
- Suggested activity: students place faces for f, g, then physically trace the pentagon paths before seeing the 3-cell frame.

12.6 Automation snippets (optional)

- A batch file that runs encoder, runner, renderer for each demo in one click.
- A ledger lint that verifies: no *_OK = 1 without *_id; column order; referential integrity of ids; storyboard hash footer equals H.

Outcome

These demos prove the pipeline end to end: clear pictures for readers, small re-checkable facts for machines, and identical artifacts on every rerun. They are small enough to teach and solid enough to audit.

13. Verification and Audit

Overview
Verification is split between internal re-checks by the small core and external confirmations by independent tools. Every green bit in run.csv must be reproducible from the trio (sigma.json, bits.map.csv, run.csv). Nothing flips to 1 without a concrete witness id when a witness is required.

13.1 Internal verification (core checker)

- Deterministic replay
    - Reconstruct the canonical bitstring s from bits.map.csv and each row's op_tag.
    - Re-run the same sparse maps and guards.
    - Refuse any nondeterminism or missing data.
- TT-IR normalization
    - Normalize 1-cells: right-associate, erase identities where allowed.
    - Normalize 2-cells and higher: serialize ordered boundaries in a fixed textual form.
    - Compute boundary_hash = sha256(normalized_boundary_string).
- Predicate re-checks
    - For each *_OK = 1, verify that the stored *_id exists and its boundary_hash matches the goal after normalization.
    - If a predicate is marked 1 without the required *_id, the ledger fails.
- Integrity sweeps
    - One-hot: no slice has two 1s.
    - Typing: all placed symbols respect src and tgt from Sigma.
    - Referential integrity: every id referenced by a row exists in the witness table and has a stable type tag.

13.2 External verification (assistants and solvers)

- Assistants (Cubical Agda, Coq, Lean)
    - Codegen a module that names witnesses by the same ids as run.csv.
    - Typecheck the module. Store a receipt with tool versions and a short digest.

    - Do not trust assistants blindly: they confirm typing, while the core still enforces normalization equality on boundaries.
- SAT/SMT
    - Export side conditions as CNF or SMT-LIB for proposed fillers or equalities.
    - On SAT, re-import the model, reconstruct a concrete witness, and re-verify internally before flipping any *_OK bit.
    - Store solver receipts: tool name, version, sat/unsat, model digest.
- Cross-tool agreement
    - When both an assistant and the core agree on a predicate, mark external_agreement=1 in a sidecar summary (optional).
    - Disagreements must not auto-flip bits; they trigger investigation and a reason code such as solver_model_mismatch or assistant_typecheck_fail.

13.3 Provenance, determinism, and hashing

- Content addressing
    - The run hash H = sha256(sigma.json || bits.map.csv || run.csv).
    - Include H in MANIFEST.json and embed H in the footer of storyboard.png and on printed nets.
- Environment capture
    - Record tool_versions in MANIFEST.json (runner, renderer, Python).
    - Optionally include OS and CPU model for reproducibility notes.
- Determinism contract
    - Given the trio, the checker must re-derive all *_OK bits and ids.
    - The renderer must produce byte-identical storyboard.png.
    - Any change to any of the trio must change H; identical H implies identical PNG bytes.

13.4 Reason-code catalog and policy

- Fixed catalog only
    - Use a closed set of reason codes (see appendix). Do not invent ad hoc strings in run.csv.
    - Examples: type_error, domain_codom_mismatch, slot_occupied, image_not_composed, nat_witness_absent, eta_missing, eps_missing, triangle_boundary_bad, pentagon_boundary_bad, waiting_on_face, no_filler_found, solver_unsat, solver_model_mismatch, nf_mismatch, witness_invalidated, display_collision, display_omission, bad_mirror.
- Fail-closed policy
    - If anything needed for a check is missing or ambiguous, leave *_OK = 0 and set a specific reason.
    - Never flip *_OK = 1 without required ids or without internal boundary verification.
- Audit strings
    - Keep free-form notes out of run.csv. If needed, write a sidecar log keyed by t and op_tag.

13.5 Audit workflow (practical)

1. Run the internal checker on the trio to confirm all *_OK bits and ids.
2. Regenerate storyboard.png and verify its bytes match the published PNG.
3. Verify MANIFEST.json, tool versions, and H.
4. Optionally verify assistant receipts and SAT/SMT receipts against the witness ids and hashes.
5. Produce an audit report: summary of green bits, any red bits with reasons, and a closed_up_to_l table for horns.

Outcome
Verification is lightweight and repeatable: a few files, a deterministic checker, and optional third-party receipts. Readers gain confidence from pictures; auditors gain confidence from small, explicit witnesses and stable hashes.

14. Evaluation Criteria

Overview
We measure progress along four axes: expressivity, constructivity, auditability, and pedagogy. Each axis has concrete, file-checkable metrics that can be computed from the trio (sigma.json, bits.map.csv, run.csv).

14.1 Expressivity

- Levels reached: max level l such that closed_up_to_l = 1 in the ledger summary.
- HIT repertoire: number and kind of higher-inductive types exercised (e.g., pushout, sphere S^n). Record as expressivity.hit[] in MANIFEST.json.
- ua coverage: count of distinct types (A,B) with ua_OK = 1; count of round-trip successes ua_rt_OK = 1.
- Horn shapes: number of distinct (n,k) pairs encountered and filled; distribution of horn_n_k_OK by level.
- Witness kinds: counts of 2-cell, 3-cell, filler_n, and any additional categories you introduce.

14.2 Constructivity

- Normalization determinism: checker recomputes all normalized boundaries and finds zero mismatches (nf_mismatch = 0).
- Round-trip rate: fraction of ua invocations with ua_rt_OK = 1.
- Horn closure rate: fraction of pending horns that ended with horn_n_k_OK = 1 vs permanent reasons.
- HIT laws: fraction of required equalities for each HIT that verified (e.g., for pushouts, po_exist_OK and po_univ_OK).
- Performance envelope: max rows in run.csv, average time per op, and memory footprint on the reference machine (document these in a README).

14.3 Auditability

- Reason coverage: every attempted check that is 0 carries a reason from the fixed catalog; no ad hoc strings in run.csv.
- Referential integrity: all referenced ids exist in the witness table; zero dangling references.

- Hash stability: same input trio regenerates the identical storyboard.png and hash H.
- External agreement: number of predicates confirmed by assistants or solvers (with receipts), and agreement rate between external tools and the core checker.
- Re-run reproducibility: independent verifier reproduces all *_OK bits and ids from the trio.

14.4 Pedagogy

- Frame density: average number of glyphs and markers per frame; keep below a target (for example, <= 12 glyphs and <= 4 markers).
- Time-to-fluency: median time for a new reader to complete the flagship demos with correct ledger outputs (measured in a small pilot).
- Error visibility: fraction of failures that produce legible display overlays or concise legend notes rather than silent ledger-only failures.
- Classroom readiness: availability of printable L/R nets with RUN.HASH and a 1-page instructor guide.
- Portability: ability to run the demos on a low-spec machine and regenerate identical artifacts.

15. Risks and Mitigations

15.1 Technical risks

- Bit growth and combinatorics: state size and N explode as Sigma grows.
  Mitigation: sparse encodings, level partitioning, compression of inactive slices, cap per-level capacities K_l, and per-demo Sigma minimization.
- Witness sprawl: too many ids to manage.
  Mitigation: content-addressed ids (hash of normalized boundary), witness tables with referential integrity checks, garbage collection of unreachable witnesses, and dedupe by boundary_hash.
- Scheduler starvation: some horns never attempted.
  Mitigation: fair round-robin by level bands, backoff with starvation alarms, and CI gates on closed_up_to_l.

- Normalization gaps: equalities fail due to missing rewrites.
  Mitigation: TT-IR golden tests, property-based tests, and an explicit rewrite catalog with versioning.

- Soundness regressions: buggy ops flip bits incorrectly.
  Mitigation: fail-closed policy, invariants rechecked every step, and double execution in CI (replay vs checker).

- Solver trust risk: external SAT/SMT returns bad models.
  Mitigation: never trust models directly; re-construct a concrete witness and re-verify internally before flipping bits.

- Schema drift: run.csv columns or order change mid-project.
  Mitigation: freeze schema per paper version; bump schema_version on any change; lints enforce exact column order.

- Performance cliffs: long runs on modest hardware.
  Mitigation: bounded search, timeouts with reason perf_budget_exceeded, caching normalized faces, and Parquet mirror (optional).

15.2 Interop and tooling risks

- Assistant mismatch: Agda/Coq/Lean library changes.
  Mitigation: version pins in MANIFEST.json; codegen targets per tool version; receipts archived.

- Renderer divergence: PNG not byte-identical across machines.
  Mitigation: deterministic fonts, seeded layout, fixed image pipeline; PNG footer includes renderer version.

- Hash instability: platform-dependent newline or CSV formatting.
  Mitigation: canonical CSV writer, LF line endings, sorted headers, explicit locale for numbers.

15.3 Pedagogy and display risks

- Display unfaithfulness: collisions or omissions.
  Mitigation: display_faithful_OK gate; render error overlays; unit tests for V_render = V \circ m \circ r.

- Cognitive load: frames too dense.
  Mitigation: density limits (<= 12 glyphs, <= 4 markers), legends with only flipped bits, and keyed overflow pages.

- Accessibility: color-only cues.
  Mitigation: shape encodings and alt text per frame.

15.4 Security, licensing, provenance

- Supply chain: tool updates inject nondeterminism.
  Mitigation: lockfiles, SBOM, and containerized runner for archival builds.
- Licensing: assistant libraries and fonts.
  Mitigation: license scan in CI; stick to permissive stacks.
- Provenance: ambiguous origin of artifacts.
  Mitigation: content addressing via H, MANIFEST with tool_versions, signed release zip.

15.5 Fail-closed policy

- Never set *_OK = 1 without required *_id.
- Any ambiguity or missing data yields a 0 with a reason code from the fixed catalog.

15.6 Recovery playbooks

- If a witness is invalidated: append a corrective row that clears dependent bits and allocates a new witness; set witness_invalidated as reason.
- If a frame fails faithfulness: keep math, render with banner, and fix V or deco in a new run.

15.7 Capacity planning (Beelink box)

- Target demo envelope: <= 50k rows in run.csv, <= 2e6 bits active across all slices, K_2 ~ 32, K_3 ~ 8 by default.
- Record actual timings and memory in README to bound expectations and guide future growth.

---

16. Related and Future Work

16.1 Related work (selected)

- Univalent Foundations and HoTT: the HoTT book and subsequent developments in Coq/Lean.
- Cubical Type Theory and Cubical Agda: computational paths and ua with judgmental computation in fragments.

- Category Theory: associativity, coherence (Mac Lane), and diagrammatic reasoning traditions.
- Higher rewriting and computational infinity-categories: algebraic presentations and normalization strategies.
- Proof engineering: small proof objects, replayable proofs, SAT/SMT integration in proof workflows.

16.2 How this differs

- We do not replace assistants; we add a small execution-and-evidence layer that emits portable proof cards (storyboard.png + run.csv) with explicit witnesses and hashes.
- The display functor is kept out of the proof loop; math is decided on the ledger, not on pixels.
- The core is intentionally tiny: sparse Boolean ops plus a TT-IR normalizer, making audit simple and re-implementation feasible.

16.3 Future directions

- Raise n: extend horns and fillers to selected 3-horns and beyond with fair scheduling.
- Expand HITs: coequalizers, spheres S^n, and other constructors with universal-property predicates.
- Monoidal structure: unitors and associators as first-class, with pentagon/triangle witnesses at scale.
- Multiple models: compare checker outputs across independent implementations.
- Stronger normalization: push more equalities into judgmental reduction to reduce witness burden.
- Proof search: guided horn filling and equivalence synthesis under resource caps.
- Pedagogy: richer classroom kits, exercises, and instructor rubrics.
- Bridges to semantics: toposes, cohesion, and synthetic infinity-geometry backends.

---

17. Epilogue - What "Higher than Voevodsky" Means Here

Not beyond in spirit, beyond in operationalization. The contribution is to make higher structure executable and auditable: explicit witnesses for each green bit, executable univalence with round-trip checks, streamed horn fillability up to a chosen level n, and two synchronized views. One is human-first (Twin Cubes frames); the other is machine-first (a small ledger of bits and ids). The contract is compact: sigma.json, bits.map.csv, run.csv, and storyboard.png, tied by a single hash H. This is not a new foundation but a practical on-ramp that lets readers gain CT/HoTT intuition quickly while giving auditors everything they need to re-check the same argument independently.

---

18. References

- Youvan, Douglas C. 2025. Twin Cubes - Executable Storyboards and Physical Manipulatives for Category Theory - Homotopy Type Theory - CT - HoTT - including source code. ResearchGate. DOI: 10.13140/RG.2.2.28641.77920.
- Univalent Foundations Program. 2013. Homotopy Type Theory: Univalent Foundations of Mathematics. Institute for Advanced Study.
- Mac Lane, Saunders. 1998. Categories for the Working Mathematician. 2nd ed. Springer.
- Licata, Dan; Brunerie, Guillaume; et al. Selected works on higher-inductive types and computational univalence.
- Cohen, Cyril; Coquand, Thierry; Huber, Simon; Mörtberg, Anders. 2018. Cubical Type Theory papers and the Cubical Agda implementation.
- The HoTT Coq Library, the Lean HoTT and mathlib efforts, and the Cubical Agda library documentation.
- Selected SAT/SMT references: Biere et al. Handbook of Satisfiability; Z3 and cvc5 system descriptions.

Appendix A - Minimal Schemas (CopySafe)

A.1 sigma.json (conceptual fields)

- objects: array of strings
- morphisms: array of {name, src, tgt}
- cells:
    - level "2": array of {name, lhs_kind, lhs_args[], rhs_kind, rhs_args[]}
    - level "3": array of {name, faces[]} where faces is an ordered list of 2-cell names
- hit: optional block for higher-inductive types (e.g., pushout)
    - pushout: {name, f, g, inl, inr, glue_param_key}
- ops: feature switches {compose: true, inverse: true, faces: true, degeneracies: true}
- version: string (freeze when published)

Example (minimal)


```
{
"objects": ["A","B"],
"morphisms": [
{"name":"id_A","src":"A","tgt":"A"},
{"name":"id_B","src":"B","tgt":"B"},
{"name":"f","src":"A","tgt":"B"},
{"name":"g","src":"B","tgt":"A"}
],
"cells": {
"2": [
{"name":"eta","lhs_kind":"morphism","lhs_args":["id_A"],"rhs_kind":"comp","rhs_args":["g","f"
]},
{"name":"eps","lhs_kind":"comp","lhs_args":["f","g"],"rhs_kind":"morphism","rhs_args":["id_B"
]}
],
"3": []
},
"hit": {},
"ops": {"compose":true,"inverse":true,"faces":true,"degeneracies":true},
"version": "sigma-1.0"
}
```

A.2 bits.map.csv (stable index map)
Header (fixed order)
idx,level,slot_kind,slot_id,symbol_kind,symbol_name,deco_key,deco_val

Row semantics

- idx: 0-based integer into s in {0,1}^N
- level: 0,1,2,3,...
- slot_kind: e.g., object_slot, edge, square, horn, filler, hit_slot
- slot_id: integer, stable within slot_kind
- symbol_kind: e.g., object, morphism, 2cell, 3cell, filler_n, hit_ctor
- symbol_name: plain string from sigma.json or constructed name
- deco_key, deco_val: optional; empty string if unused (example: cube,L)

A.3 run.csv (ledger)
Header (fixed order)
t,op_tag,bits_hex_prefix,id_OK,comp_OK,nat_OK,left_inv_OK,right_inv_OK,triangle_OK,pentagon_OK,isEquiv_f_OK,g_id,eta_id,eps_id,isEquiv_reason,ua_OK,ua_id,ua_inv_OK,ua_inv_tuple_id,ua_rt_e_OK,ua_rt_p_OK,ua_rt_OK,ua_reason,horn_n_k_OK,horn_index,n,k,filler_id,horn_reason,po_exist_OK,po_unique_OK,po_univ_OK,po_u_id,po_u_inl_id,po_u_inr_id,po_u_glue_id,po_H_id,po_reason,display_faithful_OK,display_reason

Notes

- Never set any *_OK to 1 without required *_id.
- Reasons are from a fixed catalog only (see Appendix B).

Appendix B - Reason Codes (fixed catalog)

Typing and structure

- type_error
- domain_codom_mismatch
- slot_occupied
- ungrounded_composite
- image_not_composed

Witness and predicate

- inv_missing
- eta_missing
- eps_missing
- nat_witness_absent
- triangle_boundary_bad
- pentagon_boundary_bad
- witness_invalidated
- nf_mismatch

Horns and solvers

- waiting_on_face
- no_filler_found
- duplicate_boundary
- solver_unsat
- solver_model_mismatch
- noncanonical_boundary
- scheduler_starvation
- perf_budget_exceeded

Display

- display_collision
- display_omission
- bad_mirror

Univalence

- equiv_missing
- link_mismatch
- transport_fail

- ua_boundary_bad

Pushouts and HITs

- bad_cospan
- glue_boundary_bad
- u_missing
- u_inl_bad
- u_inr_bad
- no_mediator_found

Policy

- Use only these strings in run.csv. If a new case arises, update the paper, increment schema_version, and add the new code here.

Appendix C - Abbreviations and ASCII Math

Symbols and operators (ASCII-safe)

- composition: f \circ g
- identity: id_A
- equality: =
- definitional equality (if needed): \equiv (as a literal token)
- homotopy or path equality: ~
- arrows: A -> B, A => B (use => only for meta arrows in prose)
- inverse: f^{-1}
- universe: U
- univalence maps: ua : Equiv(A,B) -> (A = B), ua^{-1} : (A = B) -> Equiv(A,B)
- naturality: G(f) \circ alpha_A = alpha_B \circ F(f)
- equivalence tuple: (g, eta: id_A ~ g \circ f, eps: f \circ g ~ id_B)
- horns and simplices: Lambda^k[n], Delta[n]
- finite sets: Fin(n)

Identifiers and file names

- display map: V_render = V \circ m \circ r
- hash: H = sha256(sigma.json || bits.map.csv || run.csv)
- files: sigma.json, bits.map.csv, run.csv, storyboard.png, MANIFEST.json

Ledger columns (canonical, snake_case)

- id_OK, comp_OK, nat_OK, left_inv_OK, right_inv_OK, triangle_OK, pentagon_OK
- isEquiv_f_OK, g_id, eta_id, eps_id, isEquiv_reason
- ua_OK, ua_id, ua_inv_OK, ua_inv_tuple_id, ua_rt_e_OK, ua_rt_p_OK, ua_rt_OK, ua_reason
- horn_n_k_OK, horn_index, n, k, filler_id, horn_reason
- po_exist_OK, po_unique_OK, po_univ_OK, po_u_id, po_u_inl_id, po_u_inr_id, po_u_glue_id, po_H_id, po_reason
- display_faithful_OK, display_reason

Appendix D - Reproducibility Checklist (copy this into your README)

Inputs frozen

- sigma.json is versioned and immutable for this paper version.
- bits.map.csv is frozen for the given sigma.json.
- run.csv is append-only; no retroactive edits.

Determinism

- Given the trio (sigma.json, bits.map.csv, run.csv), the checker recomputes all *_OK bits and ids.
- Renderer produces byte-identical storyboard.png.
- H = sha256(sigma.json || bits.map.csv || run.csv) is printed in MANIFEST.json and in the PNG footer.

Witness discipline

- Every *_OK = 1 has required *_id present.
- Every id resolves to a witness with a normalized ordered boundary and boundary_hash.

- No dangling ids.

Reasons

- Any *_OK = 0 after an attempted check has a reason from Appendix B.
- No ad hoc free-text reasons in run.csv.

Interop receipts (optional but recommended)

- Assistant receipts (tool version, files, digest) stored in a receipts folder and listed in MANIFEST optional[].
- SAT or SMT receipts (sat or unsat, model digest) stored likewise.

Display

- display_faithful_OK = 1 on all frames, or frames show an explicit banner and display_reason.

Platform notes

- Record tool_versions (runner, renderer, Python) and platform in MANIFEST.json.

Audit script

- A single script verifies schema, determinism, ids, reasons, and regenerates storyboard.png bytes for comparison.

Appendix E - Quickstart (Windows PowerShell, CopySafe)

Assumptions

- You are on Windows 11 with Python 3.11 in PATH.
- Project root: C:\Users\doug\tt

1. Create folders
   mkdir C:\Users\doug\tt{examples,out,scripts}
2. Optional venv
   py -3.11 -m venv C:\Users\doug\tt.venv
   C:\Users\doug\tt.venv\Scripts\Activate.ps1
   python -m pip install --upgrade pip
3. Place minimal files

- Save sigma.json (from Appendix A example) to C:\Users\doug\tt\examples\sigma_min.json
- Save bits.map.csv header (Appendix A.2) to C:\Users\doug\tt\out\bits.map.csv and fill with a tiny mapping suitable for the minimal sigma.
- Create an empty run.csv with the header from Appendix A.3 in C:\Users\doug\tt\out\run.csv

4. Runner skeleton (commands are illustrative)
   python -m scripts.tt_runner set_morphism --name f --src A --tgt B --run out\run.csv --map out\bits.map.csv --sigma examples\sigma_min.json
   python -m scripts.tt_runner set_morphism --name g --src B --tgt A --run out\run.csv --map out\bits.map.csv --sigma examples\sigma_min.json
   python -m scripts.tt_runner add_witness --name eta --lhs id_A --rhs "g o f" --run out\run.csv --map out\bits.map.csv --sigma examples\sigma_min.json
   python -m scripts.tt_runner add_witness --name eps --lhs "f o g" --rhs id_B --run out\run.csv --map out\bits.map.csv --sigma examples\sigma_min.json
   python -m scripts.tt_runner check_equiv --f f --g g --eta eta --eps eps --run out\run.csv --map out\bits.map.csv --sigma examples\sigma_min.json
   python -m scripts.tt_renderer storyboard --run out\run.csv --map out\bits.map.csv --sigma examples\sigma_min.json --png out\storyboard.png
5. CNF or SMT loop (optional)
   python -m scripts.tt_emit_cnf horn 2 1 out\horn_2_1.cnf
   kissat.exe out\horn_2_1.cnf > out\horn_2_1.sol
   python -m scripts.tt_import_sat out\horn_2_1.cnf out\horn_2_1.sol out\horn_2_1.catalog.json out\run.csv
6. Agda interop (optional)
   python -m scripts.tt_codegen_agda examples\sigma_min.json out\run.csv out\ProofCard\EquivUA.agda
7. Re-run determinism check

- Rebuild storyboard.png and compare bytes with the previous file.
- Verify H in MANIFEST.json matches the hash footer in the PNG.

Appendix F - Glossary (compact)

- Sigma (signature): Finite list of generators and boundaries by level (objects, morphisms, 2-cells, 3-cells, optional HITs).
- T (state tensor): One-hot occupancy over coordinates (level, slot, symbol, deco).
- s (canonical bitstring): Flattening of T into {0,1}^N using bits.map.csv order.
- bits.map.csv: Frozen index map from tensor coordinates to bit indices for s.
- run.csv (ledger): Append-only table of steps with *_OK flags, *_id fields, and reason codes.
- op_tag: Short operation label per row (e.g., set_morphism, add_witness, check_equiv).
- V_render: Display mapping V_render = V \circ m \circ r used by the viewer; read-only.
- witness: A concrete higher cell or filler with a typed, ordered boundary; carries an id and boundary_hash.
- boundary (ordered): A normalized textual form of faces around a cell (e.g., the two sides of a 2-cell, five faces of a pentagon).
- boundary_hash: sha256 of the normalized boundary string (hex, possibly truncated for display).
- TT-IR: Tiny typed intermediate representation for 0/1/2/3-cells used for normalization and hashing.
- horn: Lambda^k[n]; an n-simplex boundary with the k-th face missing.
- filler: Delta[n]; a concrete n-simplex that fills a horn, recorded with filler_id.
- closed_up_to_l: Derived boolean; 1 iff all horns with level <= l are either filled or carry permanent reasons.
- isEquiv(f): Equivalence data for f: A -> B, recorded as tuple (g, eta, eps).
- ua, ua^{-1}: Executable univalence maps between Equiv(A,B) and (A = B) in U.
- PathU: TT-IR node encoding a path p : (A = B) at the universe level.
- po_univ_OK: Boolean that records the pushout universal property (existence and uniqueness) has been witnessed.
- reason code: Fixed string explaining why a check is 0 after an attempt (see Appendix B).

- MANIFEST.json: Small file tying together artifacts, tool versions, and the content hash H.
- H (run hash): sha256(sigma.json || bits.map.csv || run.csv), also embedded in storyboard.png footer.
- runner: Program that applies ops, updates the ledger, and enforces guards.
- renderer: Program that turns the trio into storyboard.png deterministically.
- assistant: External proof assistant (Cubical Agda, Coq, Lean) used for independent typechecking of generated witnesses.
- solver: External SAT/SMT tool used to propose candidates; models are re-verified internally before acceptance.
- catalog (solver): Mapping from SAT/SMT symbols back to concrete faces or fillers for re-import and verification.

Appendix G - SMT-LIB Templates (horns, naturality, ua linkage)

G.1 One-hot horn choice (n=2, k=1 example; choose exactly one face among X1..X5)

```
; File: out/horn_2_1.smt2
(set-logic QF_UF)
; Boolean vars for candidate faces
(declare-fun X1 () Bool)
(declare-fun X2 () Bool)
(declare-fun X3 () Bool)
(declare-fun X4 () Bool)
(declare-fun X5 () Bool)

; At-least-one
(assert (or X1 X2 X3 X4 X5))
; At-most-one (pairwise; small instances)
(assert (not (and X1 X2)))
(assert (not (and X1 X3)))
(assert (not (and X1 X4)))
(assert (not (and X1 X5)))
(assert (not (and X2 X3)))
(assert (not (and X2 X4)))
(assert (not (and X2 X5)))
(assert (not (and X3 X4)))
(assert (not (and X3 X5)))
(assert (not (and X4 X5)))
```

; Optional side constraints that encode easy typing or boundary filters
; (assert <domain/codomain predicates encoded as Booleans>)

(check-sat)
(get-model)

G.2 Naturality equality as a propositional guard
; Encode a single decision: is there a 2-cell nat_w that matches
; G(f) o alpha_A = alpha_B o F(f) under the current Sigma slice.
; For a pure SAT shape, just record a witness bit NatW and re-verify internally.

(set-logic QF_UF)
(declare-fun NatW () Bool)
; At-least-one if the square was constructed (external search chooses to include)
(assert NatW)
(check-sat)
(get-model)

Note: the actual equality is re-checked by the core normalizer. The solver only selects a candidate; the importer reconstructs nat_w and verifies its ordered boundary.

G.3 ua linkage guard (optional)
; A guard that we intend to construct a path p = ua(e_tuple_hash)
; Again, selection only; real construction is internal.
(set-logic QF_UF)
(declare-fun UseUA () Bool)
(assert UseUA)
(check-sat)
(get-model)

Guidelines

- Keep SMT encodings small. Only selection bits leave the solver; the core rebuilds concrete witnesses and verifies boundaries.
- Always include (check-sat)(get-model). Re-import via Appendix P and refuse to flip bits without internal verification.
- For larger sets, replace pairwise at-most-one with a cardinality encoding library or a sequential counter; still re-verify internally.

Appendix H - Assistant Stubs (Cubical Agda, Coq, Lean)

H.1 Cubical Agda (skeleton, ASCII-safe placeholders)

```
module ProofCard.EquivUA where

open import Cubical.Core.Everything
open import Cubical.Foundations.Everything

-- Placeholders to be codegenerated from run.csv
-- A, B : U
-- f : A -> B
-- g : B -> A
-- eta : (x : A) -> Path A (x) (g (f x))
-- eps : (y : B) -> Path B (f (g y)) (y)

record Equiv (A B : Set) : Set where
field
fwd : A -> B
bwd : B -> A
leftInv : (x : A) -> Path A x (bwd (fwd x))
rightInv : (y : B) -> Path B (fwd (bwd y)) y

-- Build the equivalence from ids matched to ledger witness ids
Equiv-from-ledger : {A B : Set} -> Equiv A B
Equiv-from-ledger = record
{ fwd = f
; bwd = g
; leftInv = eta
; rightInv = eps
}

-- ua and its inverse are primitive in Cubical; here we show round-trips
ua-rt-e : Equiv-from-ledger ≡ Equiv-from-ledger
ua-rt-e = refl

-- The actual codegen should name constants by their ids from run.csv,
-- then assemble this module and let Agda typecheck it.
```

H.2 Coq (skeleton)

```
From HoTT Require Import Basics Types Equivalences.

Section EquivUA.
Context {A B : Type}.
```

```
Variable f : A -> B.
Variable g : B -> A.
Variable eta : forall x:A, x = g (f x).
Variable eps : forall y:B, f (g y) = y.

Definition equiv_from_ledger : A <~> B.
Proof.
refine (Build_Equiv A B f _).
refine (isequiv_adjointify f g _ _).
- intro y. exact (eps y).
- intro x. exact (eta x).
Defined.

(* Round-trip sketches; actual codegen can create lemmas named by ids *)
Lemma ua_rt_e : ua (equiv_from_ledger) = ua (equiv_from_ledger).
Proof. reflexivity. Qed.
End EquivUA.
```

H.3 Lean 4 (skeleton)

```
import Mathlib

namespace ProofCard

variable {A B : Type} (f : A → B) (g : B → A)
variable (eta : ∀ x : A, x = g (f x))
variable (eps : ∀ y : B, f (g y) = y)

def equivFromLedger : A ≃ B :=
{ toFun := f
, invFun := g
, left_inv := by intro x; simpa using eta x
, right_inv := by intro y; simpa using eps y
}

-- Round-trip placeholders; codegen can emit real lemmas keyed by ids.

end ProofCard
```

Notes

- These are minimal stubs. Your codegen should substitute concrete names and bodies from run.csv witness ids, then run the assistant typechecker. Store receipts separately.

Appendix I - TT-IR (syntax, normalization, equality policy)

I.1 Syntax (informal)
Types
Obj -- object names from Sigma
Hom(A,B) -- hom type between objects
Path1(f,g) -- 2-cell type (path between 1-cells)
Path2(u,v) -- 3-cell type (path between 2-cells)
U -- universe (for paths A = B)

Terms (level 1)
id A : Hom(A,A)
mor "f" : Hom(A,B) -- f declared in Sigma
comp h g f : Hom(A,D) -- right-associated h \circ g \circ f

Terms (level 2)
cell2 "eta" : Path1(lhs,rhs) -- named 2-cell
refl1 f : Path1(f,f)

Terms (level 3)
cell3 "P" : Path2(u,v) -- named 3-cell
refl2 u : Path2(u,u)

Universe paths
PathU (A,B) : U equality
Ua e : PathU (A,B) where e hashes an equivalence tuple

I.2 Serialization (for hashing)

- 1-cells: comp is printed as comp[h, g, f] with right association and no identities.
- 2-cells: print as eq(lhs_serialized, rhs_serialized).
- 3-cells: print as eq2(u_serialized, v_serialized).
- Ua: ua(e_hash) where e_hash is computed from normalized components of the tuple.
- Always use ASCII, no spaces except single commas, and fixed key ordering.

I.3 Normalization rules (selected)

- Right associate: comp(comp(h,g),f) -> comp(h,g,f).
- Erase identities: comp(f, id B) -> f; comp(id A, f) -> f, when well typed in the chosen fragment.

- Functor image (if materialized): F(comp(h,g,f)) -> comp(F h, F g, F f); F(id A) -> id F(A).
- Refl normalization: refl1 f compares by normalized f; same for refl2.
- Ua canonicalization: Ua(tuple) -> Ua(e_hash) after normalizing components.

I.4 Equality policy

- Judgmental equality: structural equality of normalized TT-IR strings.
- Propositional equality: requires a named 2-cell or 3-cell witness with ordered boundary equal (after normalization) to the goal.
- Ledger rule: set *_OK = 1 iff judgmental equality holds or a well typed witness with matching ordered boundary exists.

I.5 Boundary order

- 2-cells: boundary ordered as [lhs, rhs].
- 3-cells: boundary ordered as [f1, f2, f3, f4, f5] for the pentagon, or as defined by the specific shape.
- Horns: boundary printed as a list of (n-1)-faces in a fixed index order; hash over the exact string.

Appendix J - JSON Schemas (informal, CopySafe)

J.1 MANIFEST.json

```
{
"schema_version": "tt-ledger/1.0",
"sigma_file": "sigma.json",
"bits_map_file": "bits.map.csv",
"run_file": "run.csv",
"png_file": "storyboard.png",
"hash": "sha256:<hex>",
"tool_versions": {
"tt-runner": "x.y.z",
"renderer": "a.b.c",
"python": "3.11.6"
},
"created_utc": "2025-08-31T23:59:59Z",
"optional": [
"out/state.parquet",
```

"out/horns/horn_2_1.cnf",
"out/horns/horn_2_1.model"
],
"expressivity": { "max_level": 3, "hit": ["pushout"], "ua": true },
"environment": { "os": "Windows 11", "cpu": "Ryzen 7 5825U" }
}

Notes

- hash is H = sha256(sigma.json || bits.map.csv || run.csv) serialized exactly as bytes.
- Keep file names relative to the project root or include absolute paths if required by your archive policy.

J.2 Sigma schema (informal)

```
{
"type": "object",
"required": ["objects","morphisms","cells","ops","version"],
"properties": {
"objects": {
"type": "array",
"items": {"type":"string"},
"uniqueItems": true
},
"morphisms": {
"type": "array",
"items": {
"type":"object",
"required":["name","src","tgt"],
"properties":{
"name":{"type":"string"},
"src":{"type":"string"},
"tgt":{"type":"string"}
}
}
},
"cells": {
"type": "object",
"properties": {
"2": {
```

```
"type":"array",
"items":{
"type":"object",
"required":["name","lhs_kind","lhs_args","rhs_kind","rhs_args"],
"properties":{
"name":{"type":"string"},
"lhs_kind":{"type":"string"},
"lhs_args":{"type":"array","items":{"type":"string"}},
"rhs_kind":{"type":"string"},
"rhs_args":{"type":"array","items":{"type":"string"}}
}
}
},
"3": {
"type":"array",
"items":{
"type":"object",
"required":["name","faces"],
"properties":{
"name":{"type":"string"},
"faces":{"type":"array","items":{"type":"string"}}
}
}
}
},
"required":["2","3"]
},
"hit": {
"type":"object",
"properties":{
"pushout":{
"type":"object",
"properties":{
"name":{"type":"string"},
"f":{"type":"string"},
"g":{"type":"string"},
"inl":{"type":"string"},
"inr":{"type":"string"},
```

"glue_param_key":{"type":"string"}
}
}
},
"additionalProperties": true
},
"ops": {
"type":"object",
"properties":{
"compose":{"type":"boolean"},
"inverse":{"type":"boolean"},
"faces":{"type":"boolean"},
"degeneracies":{"type":"boolean"}
}
},
"version":{"type":"string"}
},
"additionalProperties": false
}

If this pastes cleanly, say continue and I will finish with Appendices K through P in the same CopySafe style.

Appendix K - Printable Cube Nets (spec and payload)

K.1 Files and format

- nets.pdf: two pages, Left cube (L) and Right cube (R).
- Page size: US Letter or A4; margins >= 0.25 in; print at 100 percent scale.
- Vector only (no raster text) to keep labels crisp.
- Fonts: a single sans font, 12 pt or larger on faces, 8 pt on tabs.
- Footer on each page: RUN.HASH (sha256 of sigma.json || bits.map.csv || run.csv).

K.2 Face layout and labels

- Face index order per template: U, D, L, R, F, B.
- Each face prints:

    - face code (e.g., L:F for Left cube, Front face),
    - symbol_name glyph if assigned (keep <= 10 chars),
    - optional tiny slot_id in a corner for classroom debugging.
- Limit to 12 glyphs total per cube; overflow must be omitted on paper and listed in a small legend line below the net.

K.3 QR payload (optional)

- Payload JSON (ASCII, no spaces):
  {"hash":"sha256:<hex>","sigma":"sigma.json","map":"bits.map.csv","run":"run.csv"}
- Place a small QR below the net. It is informative only; the math view derives from the trio, not the QR.

K.4 Assembly notes

- Score fold lines; cut tabs; glue in the order indicated on the template.
- Write the RUN.HASH on one flap after assembly for quick matching with digital artifacts.
- Classroom tip: provide small stickers for witness ids (e.g., eta_id, eps_id) so students can tag faces.

Appendix L - CI Snippets (determinism and lint)

L.1 Determinism check (PowerShell)

- Goal: ensure storyboard.png regenerates byte-identically from the trio.

Commands
```
$root = "C:\Users\doug\tt"
python -m scripts.tt_renderer storyboard --run "$root\out\run.csv" --map
"$root\out\bits.map.csv" --sigma "$root\examples\sigma.json" --png
"$root\out\storyboard.tmp.png"
fc /b "$root\out\storyboard.png" "$root\out\storyboard.tmp.png"
```

Expected

- fc /b reports no differences.

L.2 Ledger lint (PowerShell)

- Goal: no *_OK = 1 without required *_id, fixed columns, known reasons.

Script outline (Python, save as scripts\tt_lint.py)

```
import csv, sys
REQUIRED_COLS =
["t","op_tag","bits_hex_prefix","id_OK","comp_OK","nat_OK","left_inv_OK","right_inv_OK","tri
angle_OK","pentagon_OK","isEquiv_f_OK","g_id","eta_id","eps_id","isEquiv_reason","ua_OK","
ua_id","ua_inv_OK","ua_inv_tuple_id","ua_rt_e_OK","ua_rt_p_OK","ua_rt_OK","ua_reason","h
orn_n_k_OK","horn_index","n","k","filler_id","horn_reason","po_exist_OK","po_unique_OK","
po_univ_OK","po_u_id","po_u_inl_id","po_u_inr_id","po_u_glue_id","po_H_id","po_reason","
display_faithful_OK","display_reason"]
REASONS =
set(["type_error","domain_codom_mismatch","slot_occupied","ungrounded_composite","imag
e_not_composed","inv_missing","eta_missing","eps_missing","nat_witness_absent","triangle_
boundary_bad","pentagon_boundary_bad","witness_invalidated","nf_mismatch","waiting_on_
face","no_filler_found","duplicate_boundary","solver_unsat","solver_model_mismatch","nonca
nonical_boundary","scheduler_starvation","perf_budget_exceeded","display_collision","display
_omission","bad_mirror","equiv_missing","link_mismatch","transport_fail","ua_boundary_bad
","bad_cospan","glue_boundary_bad","u_missing","u_inl_bad","u_inr_bad","no_mediator_fou
nd"])
def ensure(cond,msg):
if not cond:
print("LINT FAIL:", msg); sys.exit(2)
with open(sys.argv[1], newline="") as f:
r = csv.DictReader(f)
ensure(r.fieldnames == REQUIRED_COLS, "column order mismatch")
for row in r:
def need_id(ok_field, id_fields):
if row[ok_field] == "1":
for k in id_fields:
if row.get(k,"") == "":
ensure(False, f"{ok_field}=1 but missing {k} at t={row['t']}")
need_id("isEquiv_f_OK", ["g_id","eta_id","eps_id"])
if row["ua_OK"] == "1": ensure(row["ua_id"]!="","ua_OK=1 but ua_id empty")
if row["ua_inv_OK"] == "1": ensure(row["ua_inv_tuple_id"]!="","ua_inv_OK=1 but
ua_inv_tuple_id empty")
if row["ua_rt_OK"] == "1": ensure(row["ua_rt_e_OK"]=="1" and
row["ua_rt_p_OK"]=="1","ua_rt_OK=1 but sub-bits not both 1")
if row["horn_n_k_OK"] == "1": ensure(row["filler_id"]!="","horn_n_k_OK=1 but filler_id
empty")
```

```
for k in ["isEquiv_reason","ua_reason","horn_reason","po_reason","display_reason"]:
v = row.get(k,"")
if v and v not in REASONS:
ensure(False, f"unknown reason {v} at t={row['t']}")
print("LINT OK")
```

CI command

```
python -m scripts.tt_lint out\run.csv
```

L.3 Hash H recomputation (Python)

- Goal: recompute sha256(sigma.json || bits.map.csv || run.csv) exactly.

Script outline (scripts\tt_hash.py)

```
import sys,hashlib
def cat_read(paths):
h = hashlib.sha256()
for p in paths:
with open(p,"rb") as f:
while True:
b = f.read(1<<16)
if not b: break
h.update(b)
return "sha256:"+h.hexdigest()
if __name__ == "__main__":
print(cat_read(sys.argv[1:]))
```

PowerShell

```
python -m scripts.tt_hash examples\sigma.json out\bits.map.csv out\run.csv
```

Compare with MANIFEST.json and the PNG footer.

Appendix M - CNF (DIMACS) Templates and Emitter

M.1 One-hot encoding template

- Vars: x1..xm represent candidate choices; exactly one must be true.

CNF fragment

```
c at-least-one
1 2 3 0
c at-most-one (pairwise)
```

-1 -2 0
-1 -3 0
-2 -3 0
...
Notes

- For large m use a sequential counter or cardinality network to avoid O(m^2) clauses. Small demos are fine with pairwise.

M.2 Naturality selector (boolean guard)

- Vars: w for "use this naturality witness".
  CNF fragment
  c choose witness
  10 0

- The importer will map var 10 back to nat_w and rebuild the 2-cell internally.

M.3 Emitter outline (Python, scripts\tt_emit_cnf.py)

```
import sys
def emit_one_hot(var_ids):
out = []
out.append(" ".join(str(v) for v in var_ids)+" 0")
for i in range(len(var_ids)):
for j in range(i+1,len(var_ids)):
out.append(f"-{var_ids[i]} -{var_ids[j]} 0")
return out
if __name__ == "__main__":
# example: write horn_2_1.cnf with 5 candidates
var_ids = [1,2,3,4,5]
lines = ["p cnf 5 "+str(1 + len(var_ids)*(len(var_ids)-1)//2)]
lines += emit_one_hot(var_ids)
open("out\horn_2_1.cnf","w").write("\n".join(lines))
```

M.4 Model format (Kissat/Glucose style)

- A solver .sol typically prints a line starting with s SATISFIABLE and a line with v assignments, e.g.:
  s SATISFIABLE
  v 1 -2 -3 -4 -5 0

M.5 Catalog for re-import

- A JSON that maps variable numbers back to candidate faces for a horn (see Appendix O).

Appendix N - Cubical Agda Demos and Codegen

N.1 Environment (Windows)

- Install Haskell Stack and Agda, or use a prebuilt Agda binary for Windows.
- Install Cubical Agda library matching your Agda version.
- Verify: agda -V

N.2 Codegen plan

- Read run.csv, collect ids for f, g, eta, eps, and any ua objects referenced.
- Generate a module ProofCard\EquivUA.agda with:
    - postulates or definitions named by ids,
    - an Equiv record assembled from those names,
    - trivial round-trip lemmas if appropriate.

N.3 Build commands (PowerShell)
cd C:\Users\doug\tt\out\ProofCard
agda -i . EquivUA.agda

N.4 Receipts

- On success, store a text receipt:
  tool=Agda, version=<x.y.z>, module=ProofCard.EquivUA, ok=1, digest=<sha256 of generated file>

N.5 Notes

- Keep generated names ASCII only.
- Avoid Unicode in the generated source so the pipeline remains CopySafe.
- Do not depend on assistant proofs to flip *_OK bits; use receipts for independent confirmation only.

Appendix O - DIMACS -> Witness Re-import (sat model to filler)

O.1 Files

- CNF: out\horn_2_1.cnf
- SOL: out\horn_2_1.sol
- Catalog: out\horn_2_1.catalog.json
- Run: out\run.csv

O.2 Catalog format (per horn)

```
{
"kind": "horn",
"n": 2,
"k": 1,
"horn_index": 17,
"vars": [
{"var": 1, "face_name": "F0", "face_kind": "2face", "face_id": 123},
{"var": 2, "face_name": "F1", "face_kind": "2face", "face_id": 124},
{"var": 3, "face_name": "F2", "face_kind": "2face", "face_id": 125},
{"var": 4, "face_name": "F3", "face_kind": "2face", "face_id": 126},
{"var": 5, "face_name": "F4", "face_kind": "2face", "face_id": 127}
],
"boundary_hash": "3c4d..."
}
```

O.3 Importer outline (Python, scripts\tt_import_sat.py)

```
import sys, re, json, csv, os
try:
from tt.runner_api import place_filler_and_verify
except Exception:
def place_filler_and_verify(n,k,horn_index, chosen_face):
return True, 10000+horn_index, ""
def parse_sol(path):
txt = open(path,"r",encoding="utf-8",errors="ignore").read()
m = re.search(r"\bv\s+(.+?)\s*0", txt, re.S)
if not m: return []
lits = m.group(1).split()
```

```
return [int(x) for x in lits]
def pick_true_var(model_lits, domain_vars):
true_vars = [v for v in domain_vars if v in model_lits]
if len(true_vars) != 1:
return None, "one_hot_violation:" + ",".join(str(v) for v in true_vars)
return true_vars[0], ""
def main():
cnf, sol, catalog_json, run_csv = sys.argv[1:5]
cat = json.load(open(catalog_json))
lits = parse_sol(sol)
domain = [e["var"] for e in cat["vars"]]
v, err = pick_true_var(lits, domain)
header =
["t","op_tag","bits_hex_prefix","horn_n_k_OK","horn_index","n","k","filler_id","horn_reason",
"import_note"]
if v is None:
row =
{"t":"","op_tag":"import_sat_horn","bits_hex_prefix":"","horn_n_k_OK":0,"horn_index":cat,"n"
:cat,"k":cat,"filler_id":"","horn_reason":err or
"solver_model_mismatch","import_note":os.path.basename(sol)}
write_row(run_csv, row, header); return
chosen = next(e for e in cat["vars"] if e["var"]==v)
ok, filler_id, reason = place_filler_and_verify(cat["n"],cat["k"],cat["horn_index"], chosen)
row = {"t":"","op_tag":"import_sat_horn","bits_hex_prefix":"","horn_n_k_OK":1 if ok else
0,"horn_index":cat,"n":cat,"k":cat,"filler_id":filler_id if ok else "","horn_reason":"" if ok else
(reason or "solver_model_mismatch"),"import_note":os.path.basename(sol)+":var="+str(v)}
write_row(run_csv, row, header)
def write_row(path,row,header):
import os
if not os.path.exists(path):
f=open(path,"w",newline="",encoding="utf-8"); w=csv.DictWriter(f,fieldnames=header);
w.writeheader(); w.writerow(row); f.close()
else:
f=open(path,"a",newline="",encoding="utf-8"); w=csv.DictWriter(f,fieldnames=header);
w.writerow(row); f.close()
if name=="main": main()
```

O.4 Usage (PowerShell)
kissat.exe out\horn_2_1.cnf > out\horn_2_1.sol
python -m scripts.tt_import_sat out\horn_2_1.cnf out\horn_2_1.sol out\horn_2_1.catalog.json out\run.csv

Appendix P - SMT-LIB -> Witness Re-import (model to filler)

P.1 Files

- SMT: out\horn_2_1.smt2 (must end with check-sat and get-model)
- Model: out\horn_2_1.model
- Catalog: out\horn_2_1.catalog.json (Boolean symbol names map to face candidates)
- Run: out\run.csv

P.2 Catalog format

```
{
"kind": "horn",
"n": 2,
"k": 1,
"horn_index": 17,
"bool_syms": [
{"sym": "X1", "face_name": "F0", "face_kind": "2face", "face_id": 123},
{"sym": "X2", "face_name": "F1", "face_kind": "2face", "face_id": 124},
{"sym": "X3", "face_name": "F2", "face_kind": "2face", "face_id": 125},
{"sym": "X4", "face_name": "F3", "face_kind": "2face", "face_id": 126},
{"sym": "X5", "face_name": "F4", "face_kind": "2face", "face_id": 127}
],
"boundary_hash": "3c4d..."
}
```

P.3 Typical model output (Z3 or cvc5)

```
sat
(model
(define-fun X1 () Bool false)
(define-fun X2 () Bool false)
(define-fun X3 () Bool true)
(define-fun X4 () Bool false)
(define-fun X5 () Bool false)
)
```

P.4 Importer outline (Python, scripts\tt_import_smt.py)

```
import sys, os, re, json, csv
try:
from tt.runner_api import place_filler_and_verify
except Exception:
def place_filler_and_verify(n,k,horn_index, chosen_face):
return True, 10000+horn_index, ""
def read_model(path):
txt = open(path,"r",encoding="utf-8",errors="ignore").read()
status = "unknown"
for s in re.findall(r"\b(sat|unsat|unknown)\b", txt):
status = s
return status, txt
def parse_bool_defs(model_text):
assigns = {}
for sym,val in re.findall(r"define-fun\s+([^\s()]+)\s+\(\s+Bool\s+(true|false))", model_text, re.I):
assigns[sym] = (val.lower()=="true")
return assigns
def pick_one_hot(cat, assigns):
domain = [e["sym"] for e in cat["bool_syms"]]
missing = [s for s in domain if s not in assigns]
if missing: return None, "model_incomplete_for_domain:"+",".join(missing)
trues = [e for e in cat["bool_syms"] if assigns.get(e["sym"],False)]
if len(trues) != 1: return None, "one_hot_violation:"+",".join(e["sym"] for e in trues)
return trues[0], ""
def write_row(run_csv, row, header):
import os
if not os.path.exists(run_csv):
f=open(run_csv,"w",newline="",encoding="utf-8"); w=csv.DictWriter(f,fieldnames=header);
w.writeheader(); w.writerow(row); f.close()
else:
f=open(run_csv,"a",newline="",encoding="utf-8"); w=csv.DictWriter(f,fieldnames=header);
w.writerow(row); f.close()
def main():
smt2_path, model_path, catalog_path, run_csv = sys.argv[1:5]
cat = json.load(open(catalog_path,"r",encoding="utf-8"))
n,k,hi = int(cat["n"]), int(cat["k"]), int(cat["horn_index"])
status, text = read_model(model_path)
```

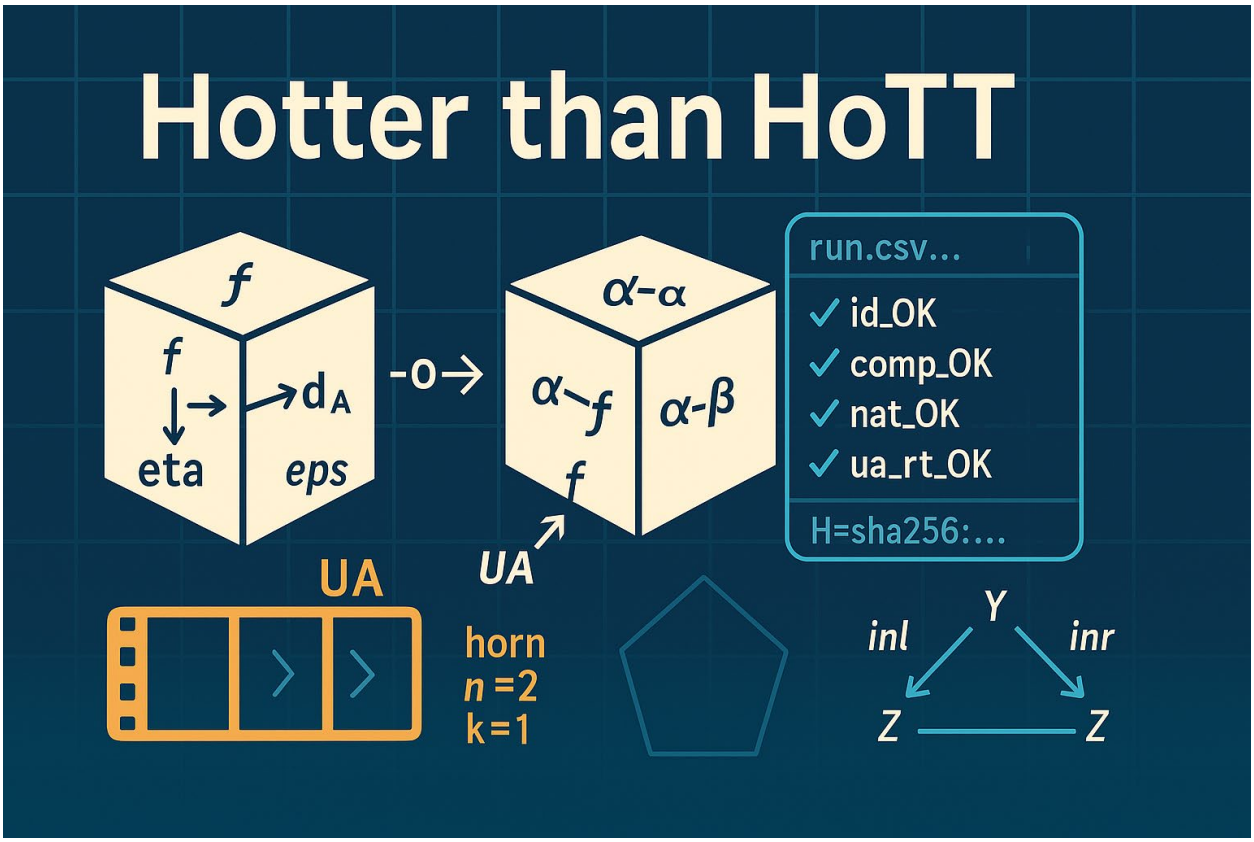
```
header =
["t","op_tag","bits_hex_prefix","horn_n_k_OK","horn_index","n","k","filler_id","horn_reason",
"import_note"]
if status == "unsat":
row =
{"t":"","op_tag":"import_smt_horn","bits_hex_prefix":"","horn_n_k_OK":0,"horn_index":hi,"n":
n,"k":k,"filler_id":"","horn_reason":"solver_unsat","import_note":os.path.basename(model_pa
th)}
write_row(run_csv,row,header); return
assigns = parse_bool_defs(text)
chosen, err = pick_one_hot(cat, assigns)
if chosen is None:
row =
{"t":"","op_tag":"import_smt_horn","bits_hex_prefix":"","horn_n_k_OK":0,"horn_index":hi,"n":
n,"k":k,"filler_id":"","horn_reason":err or
"solver_model_mismatch","import_note":os.path.basename(model_path)}
write_row(run_csv,row,header); return
ok, filler_id, reason = place_filler_and_verify(n,k,hi, chosen)
row = {"t":"","op_tag":"import_smt_horn","bits_hex_prefix":"","horn_n_k_OK":1 if ok else
0,"horn_index":hi,"n":n,"k":k,"filler_id":filler_id if ok else "","horn_reason":"" if ok else (reason
or
"solver_model_mismatch"),"import_note":os.path.basename(model_path)+":sym="+chosen["s
ym"]}
write_row(run_csv,row,header)
if __name__=="__main__": main()
```

P.5 Usage (PowerShell)

```
z3.exe C:\Users\doug\tt\out\horn_2_1.smt2 > C:\Users\doug\tt\out\horn_2_1.model
python -m scripts.tt_import_smt out\horn_2_1.smt2 out\horn_2_1.model
out\horn_2_1.catalog.json out\run.csv
```

Notes

- Always re-verify chosen fillers internally. The solver is advisory only.
- Record import rows in run.csv with horn_reason on failure and filler_id on success.


Hotter than HoTT
f
f
eta
d_A
eps
-o→
α-α
α
f
α-β
f
UA
UA
horn
n =2
k=1
run.csv…
✓ id_OK
✓ comp_OK
✓ nat_OK
✓ ua_rt_OK
H=sha256:…
Y
inl
inr
Z
Z